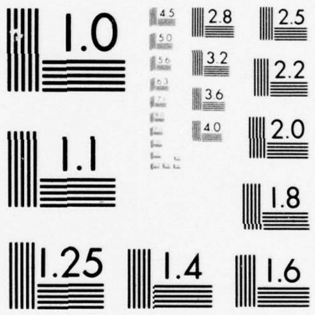
AD-A037 843 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PRODUCTION SYSTEMS AS A PROGRAMMING LANGUAGE FOR ARTIFICIAL INT--ETC(U)
DEC 76 M D RYCHENER F44620-73-C-0074

UNCLASSIFIED AFOSR-TR-77-0330-VOL-1 NL

1 OF 2
AD
A037843



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AFOSR - TR - 77 - 0330

ADA037843

8
NW

Production Systems as a Programming Language for Artificial Intelligence Applications

Volume I

Michael D. Rychener

December 1976

[illegible]

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Production Systems as a Programming Language for Artificial Intelligence Applications

Volume I

Michael D. Rychener

December 1976

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa 15213

This report reproduces a dissertation submitted to the Department of Computer Science at Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the Defense Advanced Research Projects Agency under Contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

Abstract

This thesis develops a system architecture for artificial intelligence (AI), called production systems (PSs). Each production is a simple condition-action rule, with conditions stated on a global Working Memory and actions consisting primarily of simple modifications to that memory. Actions can also consist of forming new productions. PSs have been applied to a limited extent in computer science and to a somewhat larger extent to specialized studies in AI. They are used in cognitive psychology to model human intellectual capabilities at a detailed level. With AI research tending toward larger systems with greater flexibility requirements, PSs are promising as candidates for the primary knowledge encoding medium, but certain questions and problems with PSs have been raised. The questions revolve around the practical feasibity of PSs for building large systems in a diversity of task domains, the preservation of desirable PS properties when they are applied to much larger systems than previously, and the specific advantages and disadvantages of PS architectural features.

This thesis seeks answers to such questions by constructing PSs to perform the following tasks, all of which have been developed by past AI research: extracting equations from typical high school algebra story problems (Bobrow's STUDENT); learning lists of nonsense syllable pairs (Feigenbaum's EPAM); solving a variety of puzzle tasks using a single set of general methods and processes (Newell, Shaw, Simon and Ernst's GPS); playing a simple class of chess endgames (Perdue and Berliner); discoursing in natural language about a toy blocks scene (Moran's mini-linguistic system); and solving toy blocks manipulation problems (Winograd's SHRDLU system). Each implementation is analyzed to bring out PS characteristics.

Evaluations of PSs as a programming language are made according to the traits: practical feasibility, style, degree of theory-boundness, power and overhead of expression, productivity, efficiency, architectural flexibility, and level. A taxonomy of control is presented, and measures of frequencies of usages in the PSs of various forms of control in that taxonomy are used to support the discussion of power and overhead of expression. The actual PSs are able to effectively exploit PS power in the particular areas of selections and iterations. Specific features of the particular language design used here are central to the capabilities discussed. A taxonomy of representation is developed, to provide a basis for adding openness to the PSs, replacing ad hoc internal naming conventions, and to allow measurement of the modularity of PSs, making interdependencies of various parts more examinable. The taxonomy of representation is applied to one of the larger PS programs with the finding that the split between inter-module assumptions and intra-module assumptions is roughly an order of magnitude, approximately the form of a nearly decomposable system.

PSs are found to be effective and advantageous for the programming constructs typical of AI systems. They have particular advantages in style, conciseness, and architectural flexibility. Major successes can be expected in applying PSs to large-scale understanding systems of the sort currently being explored. They are particularly useful in domains where system knowledge must grow dynamically through interaction with humans and with a task environment, but without the expense of analysis of how each new piece of knowledge must fit into existing structure. Their diversity of application and their problem-solving capabilities, both of which are deemed essential to building understanding systems, have been adequately demonstrated by this thesis.

Acknowledgments

This thesis grew out of a long series of discussions with Allen Newell, who also contributed significantly through his writings. He inspired the topic of the research, suggested methods, provided detailed criticism and encouragement of the work as it developed, and helped with the style and organization of the presentation. If significant gaps and errors remain, it is because I have not responded adequately to his objections. The other members of the thesis committee have given valuable assistance in the form of reading and commenting on the work: Raj Reddy, David Klahr, and Victor Lesser. Donald Waterman commented on some early versions of parts of the thesis and helped indirectly through his writings and through organizing a number of seminars in which production systems and representational issues were discussed. Crispin Perdue and Hans Berliner helped with Chapter V. Others who have made general contributions in ideas and in work along similar lines are Herbert Simon, Frederick Hayes-Roth, Charles Forgy, John McDermott, Kamesh Ramakrishna, Donald McCracken, Pat Langley, Thomas Moran, Charles Hedrick, Stuart Card, Lee Erman, James Gillogly, and Jack Buchanan (I apologize in advance for omissions). John McCarthy introduced me to AI in courses at Stanford, and interested me in the study of representing knowledge as rules.

My wife Pat made many things possible that might otherwise not have been, with emotional and financial support, and occasional typing and criticism. My mother and father and my wife's family have also been helpful in a number of ways.

Invaluable assistance has been rendered by the Computer Science Department at CMU as a whole, providing a rich educational and social environment, powerful computing resources, and support of the PUB document compiler and the Xerox Graphics Printer system.

Preface to Volume I

The technical report version of this thesis is split into three volumes, with Volume I containing most of the material of general interest, and with Volumes II and III presenting details of the specific studies from which the general conclusions are drawn. Thus, for all except those with serious interest in production systems, Volume I should suffice. Volume I has three chapters: the introduction, including background, motivation, and goals of the thesis (Chapter I); an introduction to production systems and to the particular language used in the remainder (Chapter II); and the conclusion, including a summary of some of the results of the other chapters (Chapter VII).

The following page gives the table of contents for the entire thesis, of which the information on Chapters I, II, and VII is pertinent to this volume. Volume II contains Chapters III and IV, and Volume III, Chapters V and VI. Each chapter has a title page, an abstract, and a detailed table of contents, which are placed directly before the first page of the chapter.

Table of Contents

For Thesis

Chapter I

Introduction

Background and Aims of Production System Research

Abstract. Production systems are a system architecture whose application to artificial intelligence has recently become attractive, due both to successes in limited domains and to results of human problem solving studies. This chapter gives some general history of the formalism and discusses some recent work that has a direct relation to the present exploration of production systems as a language for artificial intelligence. As a language, production systems are interesting, but their application to automated understanding systems, where programming features for humans are of less importance, will provide the central evaluative criteria. A priori properties and properties useful to building understanding systems are explored. General comparisons to other artificial intelligence languages are made. There is a brief discussion of the design possibilities for a production system architecture, and the peculiarities of the language to be used here are explained. Motivation is given for five tasks to be used to demonstrate and explore production system features.

Table of Contents

For Chapter I

A. Definition, History, and Approach

The endeavor at hand is to devise means to create powerful and general mechanisms, with intellectual capabilities worthy of being described as intelligent. The current view in the field of artificial intelligence (AI) is that intelligence will result when information processes of an appropriate form and content are constructed. The attempt at construction of such processes is to be complemented in AI by studying their actual and potential structure, and the structure of the information that they incorporate or might incorporate. This thesis proposes production systems as an effective tool for the task of AI.

This section has several purposes: to define what a production system is and describe abstractly how it works; to give background on the origin of the concept of production system; to sketch some important properties of the approach to be taken in experimenting with production systems; to give the general goals of the thesis; and to describe the sections to follow and the remaining chapters of the thesis.

A.1. Definition

A production system is a set of condition-action rules representing an algorithmic procedure on some domain. A rule, or production, applies to an element of the domain whenever its condition is true. The application of the production results in executing its action, producing another domain element. In using this simple view within AI, we take the domain to be the space of models of situations, represented by sets of symbolic structures. A production condition is a conjunction of schematic patterns for symbol structures, and its action is an unconditional sequence of additions, modifications, replacements, and deletions of symbol structures. Sequences of symbolic changes, resulting when productions are applied to a model, are taken to correspond to the modelled system's dynamic behavior.

The scheme just sketched hardly suffices to narrow the scope to a practical or definite computational tool. To do so requires the specification of a production system architecture. Such an architecture has four components: Working Memory, Production Memory, a recognize-act cycle, and a procedure for resolving conflicts between competing productions. Working Memory is the structure containing the dynamic knowledge state of the system, referred to above as a model of a situation. Abstractions of Working Memory elements are the primary constituents of production conditions, and manipulations of Working Memory elements are the primary constituents of production actions. Specifying the Working Memory places constraints on the attributes of its elements and on the relationships between elements. Production Memory contains all of the productions, and its specification defines allowable forms for productions and their relationships within the memory structure. Production actions usually include operators for modifying the Production Memory. The recognize-act cycle serves to control the application of productions. The usual form is that first a recognition occurs, in which a production or a set of productions is found to have its conditions satisfied with respect to the present Working Memory. The recognition usually involves matching abstract forms to specific elements. Then a selection from the recognized set is chosen for action, and the corresponding sequences of actions are performed. Performing the actions results in a new Working Memory state, and the cycle starts over with another recognition. The selection from the set of recognized productions is according to the conflict resolution principles that compose the fourth component of the architecture. These principles are usually based on the static structure of Working Memory or Production Memory, or on dynamic aspects of the system's operation such as recency of addition.

Several features of the behavior of a production system are essential. The representation of system behavior as a sequence of changing model states becomes concrete if we add the interpretation that certain of the symbol structures in the model are processed by some autonomous mechanism to result in external behavior, for instance moving a hand or making an utterance. Inputs from outside the system are somehow translated into the appropriate symbol structures and appear in the Working Memory as if they were production action manipulations, which in some architectures results in bringing them to the focus of the system's attention (which is used to resolve conflicts). Since all internal behavior is by production actions, it is through those that the overall behavior is given direction. In particular, dynamic behavior is controlled by adding Working Memory items (signals, messages, encodings of knowledge, etc.) whose intention is interpretable by other productions, often quite unrelated ones. A second means of control is by adding more productions to the system, which tends in practice to be more difficult because of longer-term effects. This is because, for reasons of history that will become clearer below, productions are considered less subject to change than Working Memory items, and in particular are rarely deleted.

Hereafter, "production system" will be abbreviated "PS", with plural form "PSs". Also, "production" will be abbreviated simply "P", with plural "Ps". The condition of a P is its left-hand side, abbreviated "LHS". Its action is its right-hand side, "RHS".

A.2. History

In this subsection, the history of PSs is used not only to provide a general basis for our approach, but also to serve as a contrast to aspects of our view of PSs. The first PSs were developed as abstract formalisms for computation, by Post (1943) and Markov (1954). Minsky's description (1967, chapter 12) is the most accessible introduction to that line of thought. Galler and Perlis (1970) started with that formal basis and proceeded to build up conventional Algolic control structures. Figure A.1 gives an example of a simple formal algorithm for reversing a string, using a PS similar to Markov's normal algorithms (adapted from Galler and Perlis, 1970, page 9). The algorithm consists of six rules, of a simple condition-action form. The list is ordered from top to bottom, with a higher rule always taking precedence over a lower one. Program control symbols are m and n, and program variables are x and y. The algorithm works on strings from some alphabet of characters, augmented for purposes of the algorithm by the program control symbols. Thus if the alphabet is (a, b, c), a legal working string is "cbnmb". Each rule consists of a string pattern (condition) followed by an arrow followed by a second string pattern (action), with the intention being to find an occurrence of the condition pattern in the working string and replace it by the action pattern. There are two special cases: if the condition pattern is empty, it always matches and the action pattern is simply appended to the left end of the working string; if the action pattern is empty, the algorithm halts. Program variables are allowed to be assigned to any symbol in the alphabet at hand, but not to the control symbols. Thus rule 4 says to find an occurrence of n (in particular, the left-most one) followed by an alphabetic symbol, and switch their order. The algorithm works by taking each character in a string and moving it across the string, placing it to the left of a previously moved character. Then it sweeps across the string removing all of the program variables. A sample execution sequence is: abc (apply rule 6) mabc (1) bmac (1) bcma (6) mbcma (1) cmbma (6) mcmbma (6) mmcmbma (2) ncmbma (4) cnmbma (3) cnbma (4) cbnma (3) cbna (4) cban (5) cba.

```
1:  mxy   -> ymx
2:  mm    -> n
3:  nm    -> n
4:  nx    -> xn
5:  n     ->
6:        -> m
```

Figure A.1 A simple Markov Algorithm for reversing a string

The present approach to PSs differs in several ways from the above. First, we take our rule sets to be unordered, for reasons of rule independence and the consequent gain in program clarity and readability. (But there have been PS architectures that used ordered rule sets.) This means that each rule must explicitly contain all the conditions on which it depends, rather than allowing rule order to implicitly set up masking conditions for rules so that a rule depends on its own conditions plus all of the conditions in the rules above it. Our PSs work with Lisp-based predicate-calculus-like assertions rather than strings of characters, for more structure and manageability. In contrast to the Galler and Perlis approach, we make no effort to build ourselves into conventional control structures, but rather leave the system open and simple, in the hope that managing control with Working Memory items allows the flexibility required for maximum intelligence.

Two other efforts to apply PS principles to conventional programming languages are exemplified by the definition of Algol 60 and by Floyd-Evans Ps for building compilers. As Minsky (1967) points out, using PSs to define languages is rather different from their use in expressing algorithms, in the sense that the former uses rules permissively and nondeterministically. That is, a definition using a set of rules uses them generatively, allowing generation of an indefinite number (usually) of grammatical language strings, but not aiming at any particular language string or subclass of strings. The algorithmic application of PSs to compiling programs has a *definite* processing aim in mind, and includes control to direct the processing to that aim. Floyd (1961) and Evans (1964) take the algorithmic approach, and their PSs are tailored to the task of parsing programming languages, incorporating, in our view, too much control, allowing subroutines of Ps, accessible by specific labels that can be the target of explicit branching commands.

PSs became part of AI research with Waterman's (1970) program, which used them to express poker heuristics in a learning task. He traces his use of PSs to both programming language research (above) and human problem solving (below). Waterman had a program structure that included other aspects besides PSs, and in fact used the conventional parts of the program to process and manipulate the Ps. His Ps were of fixed format and were used only to represent rules of thumb for making betting decisions. His program created new Ps and modified and generalized existing Ps, to achieve adaptive behavior. Our approach differs in using Ps exclusively to express entire programs, and in allowing general forms for them. The Heuristic Dendral project, with an approach similar to Waterman's in some ways, involves automatic learning of heuristics for interpreting data from chemical instruments, in particular to identify chemical compounds on the basis of occurrences of fractional substructures in mass spectrograms. Buchanan et. al. (1971, 1973) describe aspects of that work of interest to AI. Comments similar to those made on Waterman apply here. Dendral is noteworthy in being a project carried out at the frontiers of chemical research, and in achieving expertise in its area surpassing that of human experts. In conjunction with chemists, it has produced publishable new results. These and other systems are discussed further in Section E.

The body of work on which the present is most closely based is represented in papers by Newell (1972, 1973) and in the book on human problem solving by Newell and Simon (1972). Newell first applied the PS approach to narrow problem-solving contexts, extending it to perceiving and encoding processes, and in the latter succeeded in proposing models to account very closely (quantitatively) for some features of data from experiments with human subjects. The book presents more support for the usefulness of PSs in modelling human information processes, and includes a theory of human problem solving in which PSs or PS-like properties play a prominent role. The present concern is with PSs independently of the details of their use for psychological models, but their preliminary success in psychology provides important motivation for examining them seriously. Section C.2 and Section E will go into more detail on these and related topics.

A.3. Approach and goals of this thesis

The main goal of this thesis is to establish empirically that PSs are an effective language for AI applications. To date in AI, their application has been somewhat narrow: in game-playing (betting heuristics), in chemical theory formation, in learning simple linguistic rules, in representing diagnostic rules in medicine, and in a few others (see Section E). People have seemingly been reluctant to try to carry over the advantages of PSs in representing isolated heuristics within some other program framework to the construction of complete systems such as understanding systems or general problem-solving systems. In fact, there have been claims that PSs are inappropriate in a number of ways (details in Section E). There is, of course, no question that PSs are formally of sufficient power to represent arbitrary algorithms. We would like to determine instead their practical advantages and disadvantages in expressing major AI systems. This assumes that the task of building AI systems is sufficiently different from, say, writing numerical algorithms to warrant a special language system (which assumption seems to be widespread within AI). The approach to this determination is to code a half-dozen or so AI programs that have already been developed and documented and that have been sufficiently prominent to be the basis for other continuing research, whether explicit or in disguised or modified form. The PS programs themselves will support the feasibility of using PSs, and in addition will be amenable to analysis of where the power of PSs comes from, of where overhead is incurred, and of where PSs offer a richness of architectural alternatives for encoding, and of the position of PSs with respect to a few other general traits of languages. Where program listings or detailed descriptions exist for these subject programs, direct comparisons might yield valuable insights. A variety of techniques for using PSs in typical AI situations will be developed and demonstrated. Such development might then be applied to subsequent new applications. The expression of typical AI methods as PSs may result in their reformulation in interesting ways. PS implementations may have more powerful capabilities than their forebears as a direct result of their being PSs, which would point up PS characteristics in actual practice. Properties of the chosen set of AI programs might recommend them for the evaluation of other AI languages.

PSs are a remarkably simple system architecture. There are two sides to this coin. On the one hand, the familiar control and data context environment has been discarded, leaving us with only the ability to recognize patterns in a global Working Memory and to take an unconditional sequence of actions on the basis of what is recognized. Our concern as programmers is thus perhaps to try to recover some semblance of control, but it will be the case that complete control as we are accustomed to will not be necessary. That concern can also be relieved somewhat by the far goal that PS programs be written and augmented solely by automatic procedures, probably themselves written as PSs. On the other hand, there is no language bias towards any of the classical weak methods of AI, such as heuristic search (see Section D.2). Each task can be treated largely according to its peculiarities, with the building of overlying control structures as the need arises. Without going into more detail at this point, it often seems in AI research that commitment too early in a design to a particular control organization can block progress later in the design, and can in fact result in a system whose behavior is sufficient for the original aims but is increasingly resistant to extension. The present approach is to let the overlying control emerge from the structure of the task-specific knowledge, expressed as Ps, during its expansion.

Viewing the thesis as a programming task must be taken relative to two different interpretations. The first is the use of PSs by humans to encode task knowledge to form intelligent systems, which is the activity at hand. The second is the use by the intelligent systems themselves to augment their own capabilities, taking in, e.g., natural language from humans. These uses of PSs as simply another programming language and as the target for an automatic programming system undoubtedly should be evaluated according to different criteria. At present, the second view seems the more reasonable one (people tend to find it difficult to program PSs) and the one more likely to be treated adequately by the kinds of experiments proposed.

Aside from the primary goal just laid out, a number of secondary goals are present, and are attained to varying degrees according to their difficulty and to the directness of their relation to the methods used here. By encoding a number of AI tasks in a uniform notation, we seek a rational basis for AI, in terms of common program features. It may be possible to build a model of the kind of knowledge structuring that is most effectively programmed as a PS, so that further efforts with PSs could first formulate their tasks within that model, facilitating the details of the PS encoding. Such a model may or may not indicate the optimality of PSs.

Another secondary result will be more insight into the details of encoding knowledge as Ps. Some of the properties of PSs have already been elaborated, but in limited domains (see Section E), so more evidence will be useful. This will be most useful if PSs do become the target language for some automatic programming system or for an understanding system that aims at automating the acquisition of new knowledge. We may understand better what procedural knowledge is and how it is manipulated, if we study it within a model that views behavior as a series of transitions between (non-procedural) knowledge states, as sketched above in introducing PSs.

Several other secondary goals can be mentioned briefly. This thesis will establish methodological tools for making further studies along the same lines with other AI systems. This will include a set of specific benchmarks against which others could be measured. A list of desirable properties of systems will be developed, along with proper measures for them. It will provide feedback to the process of designing new PS architectures, for instance by analyzing the places where the expression of knowledge in condition patterns seems particularly clumsy. Stereotyped forms of expression might thus be made more convenient. Finally, regardless of the stated intention not to model within psychological constraints, the PSs might provide valuable input to psychological model builders. This might take the form, for instance, of pointing out places where constraints on Working Memory size might be most difficult to meet.

A.4. Overview of chapter

This chapter discusses the place of this thesis in relation to AI in general and to work on AI languages and on PSs. Section B discusses how the present work with PSs derives from more general AI goals, and how it might be considered as developing means toward those goals. Section C gives some a priori reasons why PSs look promising as an AI language, emphasizing the peculiar PS approach to long-standing AI problems and discussing the psychological motivation of using PSs. Section D gives features of some of the new AI languages, and develops comparisons of those to PSs. Section E discusses recent results from specific explorations with PSs. Section F lists the AI programs that are the subject of the body of the thesis, and sketches some of the methodology. Section G discusses some of the features of the particular PS architecture designed for this work, and makes comparisons to other PS architectures. Section H gives hints to the reader on how to find various material in the thesis while avoiding unnecessary details.

B. The Context of This Research

The goals of AI researchers belong to a diverse collection of categories, so it is necessary before going too far to understand how the present work is related to AI's major subareas. Section B.1 uses Nilsson's (1974) classification of AI areas to explain the present emphasis. Section B.2 shows how this work derives from my previous research on PSs, that is, primarily to investigate whether the scheme of analysis will carry over to wider applications of PSs, and whether conclusions on how knowledge might be automatically encoded in PSs still hold. Section B.3 explains how this thesis fits into a general strategy of studying the content knowledge required for building understanding systems.

B.1. Classification

Nilsson has divided the field of AI into a number of research areas, of which four are designated core areas, and the rest, first-level application areas (Nilsson, 1974). The four core areas are: common-sense reasoning, deduction, and problem-solving; modeling and representation of knowledge; heuristic search; and AI systems and languages. The present thesis is in the fourth category, but as a means to invesigating issues and ultimately advancing the state of the art in the first and second categories. The principal approach of interest to those is the building of understanding systems, by which I mean systems that embody knowledge about some subject area, that are able to manipulate that knowledge, including problem-solving, and that are able to exhibit communicative behavior to demonstrate their abilities and the content of the knowledge. The following argues that past research has not satisfactorily demonstrated the usefulness of various proposals for understanding systems, due to the lack of diversity of behavior, and that the present approach might therefore be more appropriate for the initiation of a large system-building effort.

A number of past efforts have dealt with various aspects of the problems encountered in building understanding systems, but have dealt mostly with the form of the design without treating a body of task-domain knowledge in sufficient quantity to show effectiveness for a large-scale system. Several lines of research that are relevant to various components of understanding systems can be mentioned:⊗ problem-solving techniques, and solving simple puzzles using means-ends analysis and heuristic search; the use of predicate-calculus notation and general uniform proof procedures; the integration of the two preceding areas for robotics problem-solving ssystems; the representation and subsequent use of structured knowledge in semantic networks; and the use of ad hoc procedures to represent knowledge, taking advantage of peculiarities of domains to avoid the costly application of uniform procedures or weak search methods. One principle that has emerged from these and others, and that will be central to the success of the PS approach, is that it is very often beneficial to add domain-specific knowledge in some form; for instance, "syntactic" methods are often considerably improved by using the "semantics"

⊗ The reader is referred to Nilsson's survey (1974) for a broader and more detailed summary of these lines of work, as well as for specific references.

of the domain. This is demonstrated in some natural language processing systems, in theorem-proving systems, and in chess programs.

In addition to the deliberate (and usually necessary) scope limitations of many of the results of AI research in the past, there are some broader respects in which the research is inadequate for attacking the larger aim of building understanding systems. The effectiveness of any single approach over a diverse set of applications has not been demonstrated. To aim at such an approach is desirable at least from the standpoint of parsimony, though parsimony might turn out to be unachievable. Even within narrow subject areas, there has been little attempt to prove comprehensiveness; for instance, in dealing with representation, each system represents its own task domain without attempting to address any of the typical examples of the others or to deal specifically with problems (representational and processing) in other approaches. This results in a set of systems covering a number of task areas but whose interactions and overlaps are quite unknown. As a result, it is difficult to tell if particular research is a real advance. Very few systems have a coherent approach to one of the primary problems of the area, the knowledge interaction problem: A particular arrangement that has proved successful on some task may become unstable when further knowledge is to be added, due to increased complexity of interaction between pieces of knowledge. This problem can be partially approached by asking how knowledge is applied when appropriate, how its appropriateness is recognized so that it can be brought to bear, and how it interacts with other knowledge in ensuring a correct result when single pieces of knowledge or single knowledge sources are insufficient by themselves.

The present work with PSs shares the characteristic of limited domain with other AI approaches, in being an intensive study of how knowledge can be expressed within the constraints of the particularly promising PS form. The underlying aim is to go on to large-scale comprehensive system building using exclusively that form. In order to make a convincing case for proceeding with PSs, therefore, we use them uniformly to achieve a wide diversity of capabilities.

There are a number of essential properties, from a conceptual standpoint, of a language or control structure, if it is to be used effectively for an understanding system. Moore and Newell (1973) give a list of dimensions on which understanding systems are to be evaluated: representation, assimilation, accommodation, action, directionality, depth of understanding, efficiency, and error. Without elaborating on the definitions of these, it can be seen that these are high-level properties of a system. For the present purposes, rather than using those traits directly, it is more useful to focus on the representation, and see how the various traits imply desirable properties for it. So the following list of primary properties of knowledge has emerged, though it is not to be put into direct correspondence with the list of eight dimensions. Knowledge within the structure should have:

Encodability - knowledge should be easily mapped from an external form to the form in the understanding system; ultimately, the encoding should be automatable.

Inspectability - content of knowledge already encoded should be readily derivable; this is the converse of encodability, and perhaps could also be called extractability.

Accessibility - knowledge should be accessible in some form for application when it is appropriate; this need not be as complete an operation as for inspection; when knowledge is accessed or applied, its own nature is not as evident as is its effect.

Operability - knowledge must be amenable to such operations as mapping, forming analogies, generalizing, optimizing, re-formulating, deducing, and inducing.

Flexibility - knowledge should have a number of alternate forms, for instance along the procedural-declarative aspect.

Organizability - there should be a variety of potential control organizations, according to the demands of various kinds of content knowledge.

Provability - there should be a way to guarantee correctness or perhaps consistency of the encoding, in some (informal) sense; this may include being able to justify the presence of some knowledge by knowing how it has been found necessary for some behavior.

These features can be seen to be reasonable if an understanding system is viewed as something that is constantly augmenting, inspecting, correcting, and applying its knowledge. They also have the property of being somehow independent of particular systems architectures, ideally being permanent and immutable properties rather than features of systems that will undergo change as we advance scientifically in their design. Thus, it is useful to present a set of secondary properties, which are more temporary or state-of-the-art dependent or even controversial, i.e., are perhaps the current set of properties that we believe are the right means to achieve the primary properties above.

Modularity - organizable in modules, each of which can be augmented independently, for the most part; in a modular organization, relations between pieces of knowledge (relations such as dependency, similarity, taking exception to, and others) are mostly within modules rather than between knowledge in distinct modules.

Uniformity - knowledge of various sorts encoded in a similar form; gains in effectiveness are expected when multiplicity of basic form is avoided.

Transparency - the representation minimally interfering with properties of what's encoded; if in encoding some knowledge, more attention must be paid to the medium or form than to the content, then transparency is lost.

Explicitness - assumptions made by knowledge should accompany it or be otherwise directly available, rather than implicit or available only after some involved computation.

Openness - avoiding coding conventions that prevent scrutiny by general processes; also, open for interaction with other knowledge, perhaps in unexpected ways and in new contexts; also, readily available when trying to diagnose errors, assign credit or blame, and other debugging operations.

Conciseness and power - expressible briefly, in manageable pieces, having significant computational effect.

Mixed procedural and declarative - expressible in a variety of ways along the active-passive dimension.

Efficiency - readily accessible in terms of computation time.

These secondary properties probably do not cover completely all aspects of the primary ones, but such a list gives us something to focus on, in terms of possible measurement, while our expertise in building understanding systems develops. Some examples of how the two sets correspond are: modularity supports encodability, organizability, inspectability, operability, and provability; openness supports operability and accessibility; conciseness, transparency, and uniformity support encodability; and mixed procedural and declarative supports flexibility. As we shall see below, some of them are obtained almost immediately from the definition of PSs, while others require testing and deliberate measurement, which activities are central to the conclusions ultimately to be drawn about PSs.

B.2. Direct precursor to this research

The present thesis will focus in part on some questions raised by recent work with a narrower focus (Rychener, 1975). A PS implementation, Studnt, of Bobrow's (1964) Stud<u>e</u>nt program, for solving high-school algebra word problems, was analyzed in detail to determine its knowledge content and to study how that knowledge corresponds to the PS representation. Knowledge was expressed as natural language statements phrased as if spoken to an (imaginary) understanding system, describing the steps to be followed and the knowledge to be applied to perform the task. The knowledge, consisting of 218 statements in natural language, was found to map onto the Ps in a many-many fashion: several pieces of knowledge per P, and several Ps using a knowledge statement in different ways. One way this comes about is the adding on of conditions to some piece of primary knowledge; the cases represented by the conditions are represented as separate Ps, with the principal piece of knowledge interacting in a number of ways, once with each qualifying condition. The mapping of knowledge to Ps was fairly direct, involving only minor amounts of programming techniques: 70% of the knowledge statements were task-domain-related, 25% were programming techniques such as knowledge about iteration and tree-structured data, and only 5% were concerned with peculiarities of PS control. For this kind of analysis, the explicit and concise character (small numbers of condition elements and actions in each P) of PSs is essential, and it is aided considerably by the fact that the Ps are an unstructured set, so that factors like lexical location do not affect how a P is to be interpreted and how its knowledge is to be determined. The Studnt program supported the assertion that PSs would be appropriate for understanding systems, as determined by the properties given at the end of Section B.1.

On the basis of the Studnt analysis, it was possible to sketch how PS programming might take place. First knowledge is formulated abstractly as a <u>problem space</u>, a representation of the possible behaviors on a problem, containing a collection of knowledge elements and operators that produce new knowledge states from current knowledge states⊗. In Studnt, for instance, the operators are actions like scanning a string of text, splitting a string, and identifying special keywords. A problem space may include plans, which specify common sequences of operator applications that lead to some desired result. In the case of Studnt, the main plan was to scan the string from left to right, and at each point, to check for dictionary tags, check arithmetic precedences, detect delimiters, and some other things, in a particular prescribed order. The problem space with plans corresponds to an <u>abstract model</u> that describes the program more precisely, and is a

⊗ The concept of problem space is discussed in more detail in Chapter VII.

more organized structuring of the problem space elements. The abstract model gives rise to a number of principal knowledge statements, which form the skeleton for the PS program. Details expressed as knowledge statements enter into interactions with the principal knowledge. An interaction can be excitatory, which results in addition of conditions to handle extra cases, inhibitory, which prevents conditions from applying, or definitional; it can deal with knowledge about when specific dynamic information is no longer necessary (*i.e., about erasing it from Working Memory*), *about specific programming* techniques, or about PS control. Defects in behavior of the PS are seen as a lack of the appropriate knowledge interactions, which were perhaps too subtle to be considered in the initial program formulation, or which are due to details of knowledge statements that weren't included in the initial set but which now are evidently needed for the problem being solved. New knowledge is stated in terms consistent with the elements of the problem space and then enters into the appropriate interactions to result in augmenting the program. In all of this augmentation, the properties of PSs prove useful: knowledge content must be extracted and examined, it must enter into interactions with other knowledge, and then it must be encoded back into the program in the appropriate places.

Several questions raised by that analysis will be followed through in this thesis. It will be determined whether the form of knowledge in Studnt is similar to its form in other PSs, and whether the analysis and its conclusions carry over. It must be investigated whether PSs can be used for certain kinds of knowledge that were not considered within the Studnt scope. And the conclusions with respect to the properties of PSs that make them appropriate for an automatic understanding and acquisition system must be re-evaluated in the light of broader evidence. The tasks chosen for this thesis were posed with these and other questions in mind, as will be discussed in Section F.

B.3. Research strategy

As mentioned in Section B.1, past efforts in AI have been concerned with exploring various segments of the problem of building understanding systems, without establishing comprehensiveness of application or of knowledge content. The present effort is similar in scope, but has as its immediately subsequent aim to push the construction to a larger scale and grapple with the *problems expected there.* This will require not only establishing PSs as an effective underlying form, but also exploring details of knowledge content. Ultimately, knowledge content probably must be explored with the following goals in mind: to see what knowledge is actually required for some specific behaviors, as opposed to what is convenient or what occurred to the first person who tried to get a running system for some task; to be able to prove that a system that understands some set of knowledge will be capable of behaving appropriately in some task domain, or in some well-described subdomain; to explore a number of alternatives and demonstrate the superiority of one approach or another, either unconditionally or varying according to subdomain; and to assign credit or blame to various pieces of knowledge for various aspects of behavior. Note that only rarely will one implementation of some AI system be sufficient to give satisfactory answers to these criteria. *Past AI systems have consistently exhibited serious* failures by these criteria, as McCarthy has pointed out in a brief review of the area, calling it the "look ma, no hands" disease (McCarthy, 1974). Exhibiting the final behavior of a system, with only a vague description of its inner workings and control principles completely obscures the search process that probably resulted in that system. Such a search probably involved intermediate systems that failed in important ways, the discovery of critical examples that forced redesign in particular directions, and the forming of key conceptual distinctions and representational advances. Seeing only the result of the search might cause a person who sets out to analyze the system's knowledge more carefully to repeat many of the same errors simply because "unexplored" alternatives appear or because the presence of some feature was not justified. The use of critical examples and test cases is a common technique in the field of linguistics (although linguists use it to debug proposals for models rather than to exercise complete working systems).

In order to systematize the study of knowledge content, it is also necessary, it seems, to have a universal way of expressing systems and their content for the purpose of comparison. PSs or a similar architecture seem, unsurprisingly, ideal for this. First, though, PSs must be demonstrated effective over a diversity of knowledge. Implementing a variety of past systems is more appropriate for this than doing a smaller study in a new task domain. At least, PSs must be shown effective for expressing knowledge, if not in efficiency of performance. In addition to putting off efficiency concerns, the present strategy also will postpone consideration of how knowledge might be automatically encoded into (learned by) a PS. If the present work elucidates what the PS would look like after acquiring certain capabilities, it will give a definite target program for a learning system to attain.

To summarize, we are engaged in building understanding systems and in exploring bases for that goal. Past efforts have elucidated disparate capabilities and tasks, but without systematizing fully the results and without using similar or inter-translatable architectural assumptions. We aim to establish PSs as a viable architecture for a number of familiar tasks, postponing questions of performance and automatic acquisition of knowledge in order to focus on analysis and evaluation.

C. The Production System Approach

This section discusses some general characteristics of PSs and shows how those are reflected in the PS representation for a variety of common procedure and data usages in AI. A principal feature of PSs is that they are neutral with respect to many recent AI language features (see Section D). There is no bias towards a particular method, e.g. heuristic search, for formulating a task. Instead the encoding can be shaped to the peculiar terms of the task. On the other hand, there are few helping features either, so that various kinds of search, for instance, have to be coded explicitly. Our far goal of using PSs to automate the encoding makes this apparent deficiency more tolerable. PSs simply encode knowledge as small, active, behavior-producing units. Knowledge is not embedded in limiting control structure, so it is potentially open and available for interaction in diverse ways. These general properties recommend PSs for use in analyzing knowledge content and systematizing AI as discussed above.

C.1. <u>How production systems might encode common structures</u>

The following summarizes how PSs are expected to be used to encode a variety of procedural structures:[*]

Ordinary control: ad hoc Working Memory data as evocation signals; symbolic goal structures or descriptions, to which P conditions can be responsive.

Selection from a set of alternatives in Working Memory: single P or set of Ps arranged in a cascade; the LHS match narrows down the set according to constraints.

Generator of possibilities to try: computation by P or coordinated set of Ps followed by some record of the generator's status, either as Ps or in Working Memory.

Decisions on control and direction of processing: sets of Ps. A stream of behavior is a sequence of such decisions made by such sets in succession, often with a single P from each set firing to represent the outcome of a decision.

Modular organization of knowledge: sets of Ps whose LHSs and RHSs share elements and that serve to elaborate various decision cases within the module.

Maintenance of local control: ordering on events (focus of attention) within a PS architecture, incorporated into the conflict resolution principles.

Planner antecedent theorems: Ps of the form event -> further action.

Planner consequent theorems: Ps of the form goal -> means proposed to achieve the goal.

Backtracking and backup in general: avoid it by making more intelligent choices, when there are real alternatives to choose from; when a choice turns out to have been bad, try to patch or update the

[*] Cf. the similar list given by Hewitt, 1971.

current state so that the process can continue from it rather than restoring some past state. (See "generator".) The global Working Memory can be used to communicate arbitrary forms of error message and other diagnostics to direct search.

A couple of things above need further explanation. The AI language Planner will be discussed further in Section D. Modularity has been treated already in Section B.1. Generation is intended for use when the elements of a set are to be examined in turn until either the full set has been processed or an element with desired properties has been found. This can be done in a variety of ways: the Ps can generate the full set of possibilities each time, with past tries eliminated (based on a record in Working Memory or on a single P that accumulates past tries, or on specific Ps that record individually previous tries, automatically set up to exclude later duplicates), and then a selection made from the remaining set, for the specific element to be output; the full set of possibilities can be computed once and stored as a P RHS, which is then inserted into Working Memory each time the generator is used, for further selection, with the RHS updated to remove the selected element; the full set of possibilities can be generated each time, to be narrowed down by previous tries stored as a single P RHS, which is updated with each new try; or some combination of the above, where, say part of the set is generated, processed element by element, then some more generated, etc.

The following summarizes similar information for data structures:

Objects (past knowledge states, dynamic problem situations, specific known world objects, etc.): in recognition form; when the object or a distinguishing part of it is in Working Memory, it is recognized, perhaps giving it a unique name so that further information can be had (if stored in other Ps) or filling in everything immediately.

Set of objects: Ps to recognize members and give the set name, and Ps to recognize the set name and give set members.

Semantic interconnections of knowledge: Ps that fire representing traversal of the arc that is the interconnection; the firing of a P makes new knowledge available.

Frames (Minsky, 1975): Ps whose "instantiation" is developed in Working Memory; a frame's default assumptions are inserted when the frame becomes active in Working Memory; later data can replace defaults in Working Memory; a frame is initially evoked according to an LHS or a set of alternative LHSs; procedures associated with a frame are just more Ps.

Specific isolated facts: RHSs of Ps, for instance of form context -> fact.

Open questions: Ps that recognize potential answers and react appropriately.

Trees of data contexts (Conniver, McDermott and Sussman, 1972): Ps that store path information so that the current state can be transformed to some desired past state, or Ps that store an entire state for direct restoration (evoked by a name for a context that is available from another source); see the comments on backup above, though, since these are an intimate feature of that control organization.

Updating past information (stored in unknown Ps): Ps that recognize an outdated fact and replace it in Working Memory, hopefully before the process that is using it gets too far with it.

C.2. Inherent properties of production systems

There are a number of properties of PSs that follow directly from their definition and from the spirit of the PS approach. Whether these can be fully exploited in large AI systems remains to be demonstrated. That is, some of the points to be raised here should be considered speculative, to be verified in practice. This subsection discusses the properties according to three different viewpoints: architectural definition, psychology, and programming.

In discussing a priori properties of the architecture, we follow a sequence of successively larger units, from condition properties and action properties through properties of the combination of Working Memory and Production Memory taken as a whole. A P condition is a pattern on Working Memory. Thus a condition might be built up by taking some set of elements from Working Memory and conjoining them, or by abstracting and generalizing on such conjunctions of elements. A condition can be seen as a selection from Working Memory of the most important features of the situation modelled and thus represents concisely the result of filtering out irrelevancies. A P action mostly performs simple modifications of Working Memory, with the most interesting properties resulting from its conventionally small size. Having unconditional sequences of actions be small means a great deal of flexibility, allowing switching quickly from, say, one approach to an alternative, and it means that processing is interruptable, since after a small number of actions, the Working Memory is again examined and in particular interrupting conditions recognized. Small size also means that the overall process can be built up incrementally, which means that pieces of the program can be left unspecified until their need comes up in actual behavior testing, by a user of a system, at which time the small number of actions needed to compensate for a missing P can be filled in (PSs are of sufficiently high level that a small number of actions accomplish a lot in terms of the overall process, but this may be a consequence of the Working Memory representation rather than of PS architectural features).

Considering condition and action together, two properties are evident. First, they are roughly equal in size usually, which is a high degree of selection for the action involved, that is, a high ratio of condition-testing per action when compared to other control architectures (see Section D). Second, the ensemble is still rather small (say a total of 10 to 15 condition and action elements), implying that the knowledge represented by the whole P is conceptually small, and in fact can be expressed as a single statement in natural language along with a few qualifying conditions (see Section B.2).

Considering relations between Ps, we have only the basic inter-communication between them using explicit data in the global Working Memory. Ps are activated by recognition of a condition in Working Memory as opposed to direct invocation, say by name, of specific Ps by other Ps. Thus a P may communicate with other Ps by making specific changes to Working Memory, but it does not know which Ps will key on those changes. This is especially true when the PS is being augmented with new methods to achieve old functions.

Focusing on features of the architecture as a whole, one property is that the dynamic transitions of Working Memory from one state to the next are quite directly represented by condition-action Ps. This is interesting from the viewpoint of taking some system's basic behavior as a sequence of transitions and then asking what parsimonious mechanism might capture it. It is also interesting from the viewpoint of asking how new Ps might arise from an existing system, the answer being that as Working Memory constantly changes, new associations between states and their successors are established. This is especially of use when the Working Memory has access to inputs from outside the system, through which changes in some external environment can be monitored and eventually described as Ps. An additional use might be to optimize existing processes by building Ps to go from one state to another with fewer actions, say by eliminating temporary control elements that are superfluous.

Overall, the complete (immediate) dynamic state of a PS is in the Working Memory, and all procedural knowledge is encoded as Ps. The full dynamic state is global and inspectable. No control context is maintained in the structure of procedures (Ps), so that each P includes everything, explicitly, that its action depends on and comprises. Within Ps, only a very small amount of context is carried over from condition to action, as bindings to pattern variables, and that context only lasts as long as the execution of the sequence of actions.

PSs can be interpreted as a model of human information processing by identifying Production Memory with human long-term memory, and Working Memory with human short-term memory. A P can be seen as a generalization of the notion of stimulus-response pair, where stimulus has been generalized to include internal symbol structures and patterns of structures, and where response has become a sequence of internal symbolic manipulations and signals associated with motor commands. The recognition part of the recognize-act cycle is considered to be accomplished very rapidly as a result of encoding P conditions in a network in which a large number of pattern-matching and element-testing operations can be carried out in parallel. Sensory perception is seen as a process that results, indirectly or indirectly, in building symbolic structures in short-term memory corresponding to perceived objects. The motor system maps short-term memory elements into the corresponding external actions. Preliminary explorations of this model, which is based on the theory of Newell and Simon (1972), indicate cycle times (full recognize-act cycles) of around 100 milliseconds, with individual actions ranging from 10 to 50 milliseconds. Additions to long-term (P) memory are thought to occur approximately every few seconds. Psychological models tend to impose constraints on various features of the architecture, such as small Working Memory (say, up to only 30 elements maximum), Working Memory that degrades over time as elements are unused by Ps (a controversial topic), limitations to the kind of pattern matching that can be done, inability to erase Ps or Working Memory items, and others. The origin of the study of PSs for AI purposes was in psychology, but factors like the computer hardware we work with has resulted in exploration of the PS design space in directions other than those dictated by psychological considerations.

With respect to programming, the primary action in augmenting a PS is to simply add Ps. Given the modular organization sketched in Section C.1, the major problem in augmenting an existing PS, in addition to forming new Ps, is to ensure that new Ps do not conflict with other Ps in the same knowledge module. (Reminder: this thesis is devoted to exploring whether the following can be realized in practice.) If a module is represented by a set of Ps, each of which makes explicit one case of how the knowledge in the module applies to a situation, then the ideal augmentation would be that new Ps would simply give more such cases. But often what is needed is to further discriminate one of the present cases, for instance splitting it into two cases according to conditions that weren't considered previously. Thus in general, it is necessary, at least locally within a module of Ps, to determine how new Ps are related to old ones. Since modules are determined by shared condition elements, the explicitness of PSs is essential in this endeavor. Augmentation is also made easier by the conciseness, high level, and small P size.

By way of summary, we can compare the properties developed in this section to the desirable properties of understanding systems as developed in Section B.1. The following gives the properties in this section that seem to provide support for the understanding system requirements.

Encodability: small unit size, explicitness of interrelations of Ps.

Inspectability: explicitness of Ps, Working Memory global.

Accessibility of knowledge: knowledge is expressed actively, evoked according to a uniform recognition procedure.

Operability of knowledge: main operations are adding Ps and elaborating P conditions and actions.

Flexibility: existence of P Memory and Working Memory as memory structures.

Organizability: P Memory has no imposed structure.

Modularity: condition-action format, explicitness.

Uniformity: Working Memory and Ps are the only representations; Ps are direct encodings of Working Memory transitions, suitably generalized.

Conciseness: small number of conditions and actions per P.

Similarity of procedures and data: condition patterns are simple generalizations of Working Memory elements, and actions specify simple changes to Working Memory.

Some of these properties cannot be verified without actually building systems, the main activity of this thesis. A better idea of PS capabilities with respect to them will emerge as the systems are built, and the finished systems will be amenable to corresponding measures.

D. General Comparisons to Other AI Languages

This section first presents some reactions to prominent features of a number of problem-solving schemes that preceded the most recent wave of innovation. The specific approach of PSs with respect to general theorem-proving systems, languages and systems for robotics, and other modelling and reasoning schemes will be discussed. Then the primary characteristics of the most recent new AI languages are reviewed and the position of PSs with respect to those characteristics is sketched.

D.1. Some reactions to older problem-solving issues

One of the oldest and most mathematically appealing approaches is to use predicate-calculus axioms to represent real-world actions and then to use uniform deductive procedures to solve problems by proving the associated mathematical theorems (see Nilsson, 1971; historically, the approach dates from the late 1950s). One reaction to this is that the uniform deductive procedures developed to date are too undirected in their search, and can't take advantage of heuristic guidance and specific shortcuts. In most problem-solving situations, specific knowledge can be applied to achieve a desirable result or to move the search in exactly the right direction, whereas a uniform deductive procedure applies more general knowledge, and is forced to iterate through a number of alternative general deductions to find an appropriate general method. Another problem is that the uniform deductive procedures tend to be unnecessarily powerful: too much can be proven, and this only serves to inundate a problem-solver with much irrelevant information and increase combinatorial explosion in exploring proof possibilities. Theorem-proving strategies that address this problem are an improvement, but remain comparatively weak.

These three intertwined issues - too much generality, too much combinatorial branching in the search, and inability to use specific heuristics effectively - have pushed some AI workers towards a procedural representation of problem-domain-specific proof strategies, for example, the early Planner formalism (Hewitt, 1969 and 1971). (A later version of Planner overcame some difficulties and will be covered by the discussion in Section D.2.) The early Planner included language primitives that allowed an exhaustive depth-first search to take place in order to explore alternatives in choices from among sets of elements and alternatives in methods for solving some problem or subproblem. An objection to Planner's form of pre-programmed proof procedures is that it is too pre-programmed and inflexible, and that it has too much action for the amount of "intelligent" selection that it does. An objection to Planner's search primitives is that still more knowledge can be explicitly applied to cut down the search and to make search that is necessary more selective. Such additional knowledge can be expressed in Planner to some extent, but the language predisposes the user to rely too much on its blind search. Relying on search where the emphasis should be on finding effective content knowledge for a task, it seems, is an error in research strategy. As argued above (Section B.3), one of the purposes for developing PSs is to establish a simple form so that content knowledge can be more freely explored.

Several other brief reactions to others' positions can be presented before summarizing how PSs may address these issues. Simon (1972) has discussed a long-standing dichotomy of approach, namely, whether to express logic in a modal form or to use a model approach. My reaction is that for convenience in exploring knowledge content required for intelligent behavior on tasks, a model approach is more direct and convenient, and that more (seemingly) complex reasoning involving beliefs, modality, potential achievability, and other issues, can be added onto a model formulation, perhaps as "epicycles". The STRIPS problem-solving system of Fikes and Nilsson (1971) uses a theorem-proving approach to applying problem operators, within a means-ends analysis searching scheme. The theorem-proving techniques, more specifically, are applied to determine whether the enabling conditions for the application of problem operators are fulfilled. I question whether such a general mechanism is appropriate for such querying of the database of assertions, and in particular would lean towards either making things more explicit in the database (Working Memory) or towards having them derivable or accessible in ways that are not as general (and potentially costly) as theorem-proving. On the first point, keeping the world of known assertions exhaustively (extensively) in the Working Memory seems more parsimonious and immediate than to assume (apparently) without proof that the full generality of theorem-proving is necessary in every problem. That is, I assume that the complexity of many domains, particularly the robotics tasks considered for STRIPS, do not warrant the general treatment, although it is useful for purely theoretical reasons to explore general formulations, especially if there is a chance that they'll prove successful. On the second point, management of the database (Working Memory), I would prefer the strategy of using task-specific storage-management "expert" procedures to determine which facts should be stored and which facts should be recomputed or rederived each time they're needed, in order to keep the database from becoming overly large. It might be best to be able to write most programs as if everything were explicit in the database, and then code a few special procedures to make the necessary adjustments. (After experience with writing specialists, perhaps more general routines could be constructed that would capture just the right set of operations.)

The following sketch of the PS approach to these issues tries to meet the above objections. PSs aim to go further in being explicit about deduction procedures than did Planner. But by using a rule format for knowledge, it is hoped that some favorable features of the pure "declarative" predicate calculus formulation can be retained. In particular, perhaps there will be retained such features as being able to use a rule in a variety of situations, to maintain generality, and to keep the processing open to adapt to task demands and to take advantage of new information, unknown when some strategy is initiated. To anticipate some conclusions of the present study of PSs, it may be possible to perform many things directly in PSs that were done previously with more powerful language features, but further it may be possible to avoid such things as heuristic search by using PSs to encode more selectivity, as determined by analysis of the content of tasks. When general theorem-proving-like techniques are needed, PSs will be used to implement them "interpretively" and potentially more intelligently, perhaps after the fashion of GPS (Newell, Shaw, and Simon, 1963) rather than relying on built-in uniform (non-interpretive) language features. Finally, it should be noted that there have been a variety of problem-solving approaches to which no strong reactions are felt and which are thus not discussed here. Many of the issues these others raise are grappled with directly in the body of the thesis.

D.2. Features of the newer AI languages

This subsection discusses briefly the major features of the most recent AI languages, drawing heavily on the tutorial survey by Bobrow and Raphael (1973). How PSs stand with respect to these features is is discussed briefly at the end of the subsection, but much more information will be presented in the context of the particular studies that are the bulk of the thesis. Features are grouped into four categories: data types, expanded control and data contexts, patterns for retrieval and invocation, and built-in indeterminate search. Each of these will be considered in turn.

The new languages that are considered to be aimed at the same applications as PSs are Planner (Hewitt, 1972), Conniver (McDermott and Sussman, 1972), QA4 (Rulifson, et al., 1972), and Popler (Davies and Julian, 1973). These are all outgrowths or extensions of list-processing languages, so that the basic data structure is an arbitrary list structure. Some of the languages have a number of additional data types such as vectors, sets, and sets with duplication. Data is stored in a common global database, and is retrieved by specifying patterns or forms to which database elements are to be matched. Procedures are evokable as a direct result of storage, retrieval, or deletion of data elements, so that various sorts of bookkeeping of the database can be set up to be done automatically. In some of the languages, the data base is so arranged that only one occurrence of a canonical form of a data element is kept. This allows the handling of certain properties at the data base level rather than by using explicit inference rules, for instance, collapsing expressions like (+ a b c) and (+ a c b) into a single element, by commutativity. Finally, programs in these languages are manipulable objects (a property inherited from the base languages), so that there is the potential to build self-modifying programs.

A second set of features revolves around the concept of allowing a program to maintain internally several versions of its data base (world), and to pass between these versions smoothly. This has undoubtedly grown out of the best-first search regime, in which a path is explored until it is no longer the most promising, at which point it is (temporarily) abandoned for some other path. A program that desires to evaluate its progress, diagnose how expectations have failed, and compare alternative explorations has a much easier time (according to the proponents of these languages) if there is an easy way to enter into any number of contexts, examine data and control status there, and resume execution from wherever it chooses. The most coherent and efficient implementation of this concept involves the "spaghetti stack" organization (Bobrow and Wegbreit, 1973). Another motivation for separating so distinctly the various contexts is to allow the processing to be carried out in a multi-processing computer environment, in which a number of alternative branches in a search tree could be explored in parallel.

Pattern-matching provides the nucleus of a third set of features. It is possible to specify, for retrieval purposes, matches on complex symbol structures, with new structures built on the basis of match success. The data-base procedures mentioned above are all based on sensitivity to patterns, that is, are keyed to classes of data elements as specified by patterns. Pattern-matching provides a very powerful way to evoke more general procedures. A procedure can be indexed according to the form of result that it achieves, and whenever that result is desired by other procedures, it is evoked, either automatically or after passing further constraint testing.

The fourth and final set of features deals with built-in search mechanisms and with concise ways of expressing the non-determinism that gives rise to search. This concept is closely related to the second set of features, in that a choice-point in a search gives rise to a subdivision in the current data and control context. Similarly, it can be seen as a device to exploit parallelism in computer hardware with a minimum burden on the user to coordinate various processes. Often programs can be written as if no choices had been made, that is, the search mechanics and the intricacies of alternative data and control contexts are essentially invisible. A variant on the invisibility exists in languages that allow the user to manipulate the possibilities, with the facility of ordering the search according to user-defined priorities.

How do PSs stand with respect to these features? The Working Memory of a PS corresponds directly to a database, but currently no PSs have made the leap to the variety of data types that is available in some of these other languages, remaining in the basic list structure domain. The current (consensus) PS approach is not to view Working Memory as extendable to a tree of dynamic data contexts, in keeping with the PS approach to search, which will be discussed immediately below. Pattern-matching is an essential part of the recognition of P conditions, so PSs are in line with the above features in the third category.

With respect to search, especially built-in search mechanisms, PSs take a divergent position. Search using a PS must exploit the extra power available in PSs' condition-recognition capability. Patterns as expressed in LHSs of Ps tend to be much more complex than, say, evocation conditions for procedures in the other languages. The PS approach is thus to use selectivity in choosing a direction for search, so that ideally search is avoided altogether and the right choice made initially. (In theory, there is nothing to prevent PSs from being embedded within some scheme by which alternative database contexts would be kept, with a set of RHS primitives provided for switching between them.) For doing basic database bookkeeping, Ps themselves are probably effective without further mechanisms along the lines of the special database procedures described above. (Again, though, nothing prevents such additions, if an application should warrant it.) For search processes investigated in this thesis, the aim is to use PSs to encode what's needed explicitly, and if that turns out to be burdensome or clumsy or too large a proportion of the problem-solving, to then propose more specialized mechanisms - but the expectation is that no such characteristics will be observed.

There should be no problems within current PS mechanisms in achieving the main functions of trees of data and control contexts: communication of success and more importantly of failure and reasons for failure; access to suspended search states; redirection of the search to more promising alternatives; and application of parallel processing. Communication is more a problem of representation than of control structure, though perhaps less control context to interfere will prove to be an advantage. With respect to suspended search states, Ps can be used to store state information or path information from which a state can be reconstructed, putting the information out of the way of current processing until required. Redirection is more a problem of building a symbolic description of the alternatives and comparing them than of control structure. The recognition step in the recognize-act cycle can use parallelism, while it seems best on the basis of human problem solving to retain seriality of the actions of Ps. Although these are just carefully considered expectations, it is anticipated that actual problems in the body of the thesis will illustrate PS capabilities along these lines.

It should be pointed out that PSs tend to have a depth-first search orientation, provided that the way of resolving conflicts between Ps favors those Ps that treat more recent Working Memory elements. As a PS is processing, current "goals" give rise to new information which will temporarily take precedence over information associated with other goals. Such a "pushing down in the stack" can occur a number of times, until a point is reached where the most recent data has been processed fully, at which point control would fall back according to the conflict resolution, to consider slightly older data. Further pushings and subsequent falling back would eventually get back to the goal that initiated the sequence. The resulting behavior is easily seen to correspond to depth-first search.

Even though the basis of PSs is pattern-matching similar in form to that in other AI languages, the control and use of match results is distinctive. In PSs, short sequences of unconditional actions are constantly alternating with matching that is generally more complex than in the others. This should bring more flexibility, make shifting directions easier, and allow processing to be more easily interruptable as new information appears. PSs encode knowledge more uniformly, and PS languages tend to be much simpler on the surface than the others, but without sacrificing power or conciseness. There is little static ordering between distinct P condition patterns, and choices are made uniformly on the basis of the conflict resolution principles. Other languages build rather rigid structures of patterns, for instance putting them together in subroutines or nesting them dynamically with shared variable bindings and control primitives. The evocation of procedures as patterns of data emerge in Working Memory seems more open in PSs because there is no way to evoke procedures more directly, by name - the only recourse for a process to evoke others is through global communication, and a P that sets up a goal can make no assumptions about which process will attempt to achieve it. The only local, hidden context is in variable bindings within Ps, and that lasts only for the duration of the P's action sequence.

E. Direct Antecedents and Relatives of the Present Approach

This section discusses a variety of work that can be considered as directly related to the present research. There is a rough grouping of research into work that has been well-known for a few years, work that is current but whose approach differs somewhat from the present one, and work that is along similar lines to the present. The first group includes some of the bases for PSs in specialized programming languages that are not PSs and a few pioneering efforts that brought PSs to the attention of current AI researchers. The second group includes applications of PS principles in varying degrees to rule induction, medical diagnosis, and speech understanding. The third group includes a number of psychological models, encompassing problem-solving, visual imagery, primitive perceptual and quantitative processes, and computer programming. It also includes work on serial pattern acquisition, simple association learning, and a detailed analysis of an implementation of a classic AI program. The aim in presenting this survey is to raise a number of issues, examine failings and open questions, and treat the differences of approach that are represented.

The first programming language to incorporate PS ideas was Comit (Yngve, 1962). Comit specialized in recognizing patterns of words within lists of words, associating with each pattern a manipulation of the word list matched by the pattern. Rules consisting of a pattern followed by manipulations were organized into named subroutines, within which rules were tested in a specific order. Data structures to which patterns were matched were also named and were subject to reorganization by commands within rules. Yngve stated quick programmability, with satisfactory efficiency, as properties of Comit when applied to information retrieval tasks. Bobrow (1964) applied a variant of Comit, Meteor, to good effect in building an AI system, Student, for solving high school algebra word problems. He pointed out that the language was easy to read and write programs in, and that the class of problems handled by the system could be easily extended by adding syntactic rules to the program. (The actual linear equations were solved by a Lisp subprocedure.) Neither Yngve nor Bobrow apparently realized the architectural possibilities that have come to light since then, as discussed in this chapter.

Comit fell out of use, probably due to the appearance of the more versatile Snobol language (Griswold, et al., 1968), which is presently in widespread use. Snobol uses many of the same features as Comit, but is less pure in PS terms because of the inclusion of a number of features of more conventional programming languages - not all of the statements do pattern matching to strings of characters, and the use of program variables is less cumbersome than in Comit. Snobol has a character basis, as opposed to Comit's word (Lisp atom) basis. The recent Lisp70 programming language (Tesler et al., 1973, and Enea and Colby, 1973) revives the List-processing basis as in Comit but uses still more powerful features for overall control. Lisp70 has basic units composed of rules that match to an input "stream" and perform basic rewriting actions on that stream. It has been applied to finding patterns in natural language inputs, on which to base responses in a dialog, and to planning tasks in robotics. In addition to the pattern-directed aspect, Lisp70 aims to include such mechanisms as coroutining, backtracking, use of long-term database memory, and language extensibility (to be achieved easily within the rule structure).

To summarize on the PS-like character of some programming languages, it is clear that the power of the basic operation of matching followed by action has been realized in a number of ways. The mentioned approaches, however, have sacrificed a number of the advantages of PSs for building understanding systems, by yielding to the tendency to embed the nechanism in a framework not unlike more conventional languages (a tendency that is generally followed by special-purpose languages). Our approach is to try to maintain the pure PS architecture as a viable alternative.

Turning to areas within AI where PS principles have been applied, Siklossy (1968) used a rule format to express acquired knowledge in a program for learning natural language, in particular learning to generate language from language-independent functional expressions. His P-like rules matched elements of the structured functional language using only tests for set membership (as opposed to pattern variables), and performed a translation and rearrangement of the matched elements to produce natural language strings (Russian and German). The program initialized its knowledge with a pair of language strings that were chosen to be definitive in a particular way, and then proceeded to augment its set of Ps by attempting to extend the performance to other natural language strings. Siklossy observed some dependence of program behavior on the ordering of elements in its training sequence. The program was able to use the PS representation to allow newly-added rules to incorporate intelligent guesses and to avoid errors of certain types in advance, as opposed to necessitating a process of error recovery. One PS-architectural consideration he raised is that he started out thinking a strictly-ordered list of Ps, with the most recently-added ones taking precedence over older ones, would suffice for his task. He later relaxed that ordering so that several rules could be matched simultaneously, allowing the best match from the set to be used in further processing - the "best" is in terms of properties of the translations produced by the different rules. Siklossy's program was successful on a limited set of utterances, and he gave no discouragement to extending it, but no one has taken up the challenge.

For Siklossy, the P rules were a small, augmentable part of the system, with other major mechanisms encoded directly in a list-processing language, and the same is true for Waterman's (1970) program for learning betting heuristics for Poker. The Poker PS used strict linear ordering for conflict resolution, allowing a new rule to mask out the action of a previous rule. The patterns matched by the Ps were based on values of a number of heuristic dimensions, pre-defined as essential to betting, and the action of each rule is a single betting decision (raise, call, etc.). The poker program converged fairly rapidly to a level of skill above the average amateur.

A third program of research that uses PSs as an augmentable subsystem is the Dendral program of Buchanan, et al. (1971, 1973). It attempts a much more ambitious task, and one whose application to practical science is immediate: the analysis of chemical molecules and the building of a theory of that analysis. The chemistry involves postulating processes of molecule fragmentation that show up as measurable quantities of various known simpler molecules, which quantities are then used for the analysis. Dendral is really two programs: Heuristic Dendral uses a set of rules directly for the analysis; Meta-Dendral, the more difficult and developmental part, builds the theory, representing it as rules usable by the heuristic part, from more primitive, directly observable data. Meta-Dendral must first search in a space of possible process rules, given the behavior under fragmentation of known molecules, expressed as input-output pairs. The rules from this first step are then subjected to processes of aggregation and generalization, to try to get a coherent and parsimonious set of rules. The aggregation of the rules is in two steps, one based on similarities in the processes involved (RHSs of rules) and the other based on trying to get more abstract descriptions of classes of molecules (LHSs of rules). The process that builds and manipulates the rule set is quite specialized and takes advantage of chemistry knowledge. Specific data on its computational behavior are not known (it almost certainly is not forming new rules interactively in real time), but its results are publishable, on a par with those of human specialists in the area. The main emphasis on current Dendral research is on construction of rules rather than on processing comparable to the focus of the present work. That is, rules seem not to be processed in an immediate, recognize-act cycle, but rather in an inductive-explanatory mode. The principal contribution to PS research is in the basic representation of knowledge and in the processing that automatically produces elements in that representation.

Now we turn to a discussion of more current work, consisting in part of theoretical designs and in part of preliminary, promising results. Becker (1973) describes a PS-like model of what he calls intermediate-level cognition: something between low-level acts such as moving a hand to a location and high-level acts such as proving a theorem. This intermediate level is meant to encompass most of the commonplace acts that proceed in humans at a level just below what we are aware of, and in a non-intentional manner. Becker's model, which is described as if largely unverified by experiments, takes a stream of sensory data and motor actions, and transforms that stream into a set of situation-action-result rules. The stream is partitioned, usually at salient features such as the experience of a rewarding sensation or the fulfillment of some goal. A set of such approximate rules, some of which will contain irrelevant factors and erroneous (non-general) associations, is refined through further experience, which includes deliberate attempts by the modelled system to achieve repetitions of rewarding states. The refinement consists of adjusting numerical weightings associated with rule elements, both with respect to individual elements' presence and overall rule validity. Rules are used both in a recognize-act mode and in a goal-seeking mode, in which an attempt is made to fulfill an LHS corresponding to an RHS that contains a necessary element of a goal state. The primary contribution of the model for present PS work is the idea that Ps may be constructable directly from patterns of changing Working Memory states, certainly a scheme that would parsimoniously carry out a general sort of knowledge acquisition.

Hedrick (1974) uses PSs to try to synthesize and extend work in several diverse AI task areas. He exhibits a PS scheme that is applicable to learning to recognize natural language utterances, and produce a semantic representation for it, and to inducing serial patterns such as are common in intelligence tests. (The system was designed to encompass several other tasks as well.) Learning in both cases takes place as the system is presented with examples of input-output pairs. The existing PS is applied to each new example, and if its behavior is incorrect, adjustment procedures are applied to augment the PS and make changes to existing Ps. As more examples are seen, Ps are generalized by making constants into pattern variables, and by refining semantic relations that are tested in the recognition step. The program determines which changes to make by a "dynamic analysis", a search through a space of possible results, applying measures to reject changes that are not the most parsimonious. If it is decided to add a new rule to the PS, a "static analysis" is applied to determine the kinds of relations to include in the P condition. The static analysis and the P conditions themselves make use of a semantic network that holds such information as "A NEXT B", "B NEXT C", "JOHN ISA MAN", and "DOG ISA

ANIMAL". Thus if two elements are suspected to be relevant to the condition of a P, the semantic net can be searched to find some relation that holds between them. When found, it can be added to make the P more exact and also to make it subject to being generalized into forms that retain some semantic content. The semantic network is kept as a long-term memory in addition to, and quite distinct from, P Memory. As in some of the systems mentioned above, Hedrick's Ps are not applied to an input according to a recognize-act cycle, but are used in a bottom-up parsing mode, involving a search among rule-application sequences. The entire system isn't uniformly encoded as Ps, but only the small augmentable part of the system is, as has also been encountered above. The primary problems raised by his work are the challenge to represent and effectively use semantic-network-like knowledge, without going beyond the basic PS architecture, and to augment a PS without going through the expensive and combinatorial searching involved in his dynamic analysis. To be fair, he did propose some approaches to solving his combinatorial problems, but they are still not at all within the bounds of the spirit of our approach (to be summarized at the end of this section). His work has raised, therefore, interesting challenges, in addition to unifying and exploring the task domain.⊗

Erman and Lesser (1975) present a system organization that has a number of traits in common with a PS approach. The aim of their system is to understand speech by allowing a number of knowledge specialists to work on an utterance cooperatively through a global "blackboard". Each specialist contributes only where its area of expertise is applicable and without knowing how that contribution might interact with the workings of the others. The blackboard contains partial working hypotheses encoded in a form familiar to all of the specialists. A focus of attention is maintained so that computing effort can be allocated among the various knowledge sources, to drive the global process towards an acceptable complete hypothesis. It differs from a PS in that each knowledge source is a relatively large program, with evocation controlled according to the result of executing a somewhat smaller pre-condition program. Thus action is in much bigger pieces of unconditional execution (at least, unconditional with respect to global effort allocation), and conflict resolution can know less about the internal structure of the knowledge sources in making its decision about where to allocate computing effort. This work (still in progress) can potentially contribute to our knowledge of PSs by developing task-independent heuristics for making decisions about how to decide between conflicting sources, and by exploring the consequences of using the global communication memory.

The MYCIN system (Shortliffe, 1974, and Davis, Buchanan, and Shortliffe, 1975) is a successful use of PSs to represent knowledge for medical diagnosis. It takes advantage of a fortuitous correspondence between the expressive level of PSs and the way physicians express (or can easily learn to express) their diagnostic knowledge. Rules take the general form "premises -> conclusions", and are usually used to reason and chain backwards, rather than being executed in the forward, recognize-act manner (the latter is used in a few exceptional cases).⊗⊗ This means that if the program wants to conclude

⊗ Hedrick's work is also closely related to certain pattern-recognition and concept-formation approaches to learning, which are developing into areas that may soon benefit PS research; a recent paper is Hayes-Roth and McDermott (1976).
⊗⊗ Cf. consequent theorems in Hewitt's Planner (1969); Anderson and Gillogly (1976) are applying MYCIN-like rules to the building of interfaces between users and complex systems; also, backward chaining using a rule-like formalism for reasoning in the construction of Algol-like robotics programs is described by Buchanan (1974).

something, it tries to find out what it can about the premises in the LHSs of rules that contain the desired conclusion in their RHSs. This "finding out" can involve interactively gathering experimental data from a human informant, it can be done by further backward chaining, using other Ps, or it can be computed by some internal function. Additional refinement is obtained by using confidence ratings on the various Ps, essentially stating the confidence in the rule by the rule informant (an expert physician). (This domain is characterized by few certainties.) These values can be thought of as probabilities, although they are not combined, when a conclusion is the result of a number of rule applications, according to conventional formulas of probability theory. (One of Shortliffe's biggest problems was to determine, empirically, an appropriate combination mechanism.)

Davis et al. point out several features attributable to PSs that are essential to their effort, and several problems that are obtained as undesirable accompaniment to positive-seeming features. As already mentioned, PSs are close to how users want to express their knowledge, so that the process of acquisition by the system is direct. Ps are found to be easily read and easily composed. Since PSs are so close to a natural expression of the user's knowledge, often the program's behavior can be explained by displaying the rule or rules it is working with. This is used when an error occurs or when the user is not sure of the reasons behind a query from the system. The program can answer "why" questions by giving the higher-up rules (supergoals), which evoked the rule being examined, and can answer "how" questions by indicating lower-down (subgoal) rules, which are about to be evoked in continuing processing. One of the main problems with acquisition of new rules is to ensure that they are not directly contradictory to existing rules, a problem exacerbated by the lack of exact theory to evaluate the meanings of the confidence levels assigned to Ps. Another problem is to make sure that a new rule takes into account all of the premises that other rules have used in similar circumstances, but this is alleviated by taking advantages of rule similarities (within program-determined classes of rules), to allow the user to be reminded of possible omissions. In a similar vein, it is sometimes the case that updating some data structure, for instance a set of values that some parameter might take on, requires a number of related changes to other Ps, raising the question of whether it might not be possible and more useful to have such structures expressed once, globally, rather than distributed through the Ps as assumptions about the premises being tested. A final feature that was added to give the rule system more direction in its backward chaining is the concept of meta-rules, rules whose conditions refer to the kinds of conditions that other rules are testing. This allows ready expression of heuristics that prefer one set of rules to another, for a particular problem. It gets around some of the problems of control that the MYCIN group have with their rules, and, since meta-rules can be stated for meta-rules themselves, opens up the process for even higher strategic guidance. Full consequences of the use of these meta-rules on the overall computational characteristics have not yet been explored, though. To summarize, the MYCIN research elucidates some techniques for using PSs in a goal-chaining fashion, makes bold advances into the realm of using numerical weights on rules, and raises issues with respect to the design of PSs to take advantage of possible automatic explanatory capabilities. Care must be taken, however, not to expect the effects of the fortuitous fit of the formalism to the domain to be present elsewhere.

While other current PS work raises a number of important issues and makes the tasks in a number of areas more clearcut, the present thesis aims to continue a line of work represented by a number of reports to be discussed now. Newell's research on PSs

(1967, 1972, 1973, and Newell and Simon, 1965, 1972) is the basis for most of the others. He introduced PSs in the general domain of modeling problem solving, and extended it to a number of other tasks. A PS for cryptarithmetic problems, involving finding digits to be substituted for letters in simple addition problems, emphasized the use of explicit goals to achieve control. In the process of developing that, there emerged a number of important characteristics of PSs, many of which have been incorporated into the discussion in this chapter. A PS for perceptual encoding was sketched in connection with a task of grouping objects into describable categories. In the 1973 paper, he used a series of related PSs to model tasks in the Sternberg paradigm, in which a human subject is given a set of digits in rapid succession and then asked whether a particular digit is in that set. This Sternberg task was used to illustrate the use of PSs for a very detailed fit of a model to actual timing characteristics of humans. The particular theory of how PSs fit into human problem solving, as sketched in Section C.2, is given in full in Newell and Simon (1972).

Several new points with respect to PSs that Newell makes in his explorations can be brought up here. He notes that in many cases it is not easy to arrange the PS to produce the desired behavior, a difficulty that seems unavoidably tied to the favorable characteristics of PSs. In particular there is the problem of maintaining local control (unexpected Ps fire) and of avoiding unwanted side effects (interference with others' global Working Memory assumptions). There is a certain freedom of programming in PSs, in that it is possible to construct a wide range of them to achieve a single task with a variety of execution characteristics. For instance, they can be readily used to represent an evolutionary sequence of system behaviors. In modeling characteristic memory unreliability in humans, Newell notes that PSs offer a mechanism of coupling, which can be used to increase reliability. Coupling is increased by strengthening the interrelationship of the outputs of one P with the inputs of another.

Klahr (1973) has also used PSs to achieve precise fits of a model to reaction-time data taken in experiments with humans. He used small PSs to model aspects of counting and addition. His evaluation of PSs is that they are much to be preferred over other common techniques such as flow diagrams, both in their basis in theory and in their precision, but that they are rather difficult to construct. In a more recent paper (Klahr, 1976), he discusses PSs for seriation, conservation, and quantification in children, and discusses general issues of cognitive development. He points out the usefulness of PSs as models of specific performance within a developmental sequence, but leaves open whether PSs or any known organization can plausibly model a complete sequence.

Young (1973) used PSs to build a flexible model of various stages of seriation behavior in children. His seriation task involved having a child arrange a set of clearly-distinguishable blocks in order of size, a task which children perform with varying degrees of proficiency at different ages. He demonstrated the flexibility of composition of his PSs by putting together a "kit" of PSs, from which varying subsets could be chosen to result in the various stages of performance of the task. Five aspects of Young's work are of interest here. His Ps were locally plausible, with each P taken by itself containing something reasonable with respect to the task domain. He noted in different experimental modes that the Ps were able to handle effectively the task variants. PSs are able to adapt to task demands without deliberate evocation of an "adaptive" process, that is, by making use of their inherent recognize-act nature. His Ps most decidedly represented skill (a direct encoding of what a child does) rather than knowledge (what a child knows, a form that only weakly says what he does as a result), which position he contrasted with other psychological models that represent knowledge rather than skill, and he expressed hope for a synthesis or middle ground. Finally, he proposed a mechanism through which development might take place: as a child repeatedly executes various manipulations in the external world, his cognitive system acquires Ps that anticipate the results of those actions (making use of the time in which motor actions leave his cognitive system free of processing demands), and that eventually begin to take part in planning and mistake-avoiding thought.

Moran (1973) and Farley (1974) both used a similar PS architecture to model human behavior on visual-imagery tasks. Though the tasks were quite distinct, they both involved using a PS to organize encoded visual inputs into known geometric shapes, and to use the processed encodings further in the tasks, e.g., to anticipate more properties of the visual environment. Moran made several points about PSs that contribute to this discussion. He organized his Ps into subroutines, in which control could be localized, but he realized this violated the spirit of PSs. He also admitted to using special tags in Working Memory to achieve obscure kinds of control and communication between Ps. His control problems led him to conclude that means should be devised, to be expressable within Ps, in order to make control more rational without losing other advantages of PSs.

Brooks (1975) used PSs to model some immediately observable processes in writing code in a programming language. His PS started out with detailed plans of how a program was to be written, and proceeded to fill in the details and produce the program code. His model is not a pure PS, but makes heavy use of operators coded in his base language, Lisp. His Ps are very specific to pieces of the plans, and tend to represent fairly large program steps, much larger that Ps in most of the other models discussed here - on the order of seconds of human thinking time as opposed to tenths of seconds. This is probably due to not coding the entire process as a PS and to weaknesses in his Ps' representational power and pattern-matching capabilities. That is, if his PS were forced to grapple with a number of the more basic operations (rather than using Lisp) and if it were able to express more general pattern matches (for instance, he has no pattern variables for comparing results from one element match to another), his Ps would tend to be broken down into simpler units with more use made of intermediate representational and control elements. He found it effective to use PSs to express the general coding strategy, used by his experimental subject, of writing some code and if necessary making patches to it later (as opposed to, say, a backtracking search through possible program modifications).

Waterman (1974, 1975) focused on self-modifying PSs for several tasks: basic arithmetic, verbal learning (Feigenbaum's 1963 EPAM), and series completion. He achieved some impressive behaviors from systems consisting of an ordered list of Ps by the simple operation of adding Ps to the list at judicious locations. He found that PSs are concise and powerful, and that there are advantages to using a uniform notation for the fixed and growing parts of learning systems. The challenge he presents is to achieve similar results without using the psychologically implausible ordered-PS architecture.⊗

We can now summarize the conclusions that can be drawn from the survey just

⊗ Actually, some consider local (i.e. on a few Ps) ordering plausible, and a total memory ordering implausible.

presented, and re-emphasize a number of questions that have been raised by other PS workers. First, it should be clear that while a number of systems have performed difficult tasks from an AI viewpoint, there has been no honest attempt to explore the consequences of using PSs over a diversity of AI tasks. Such an endeavor is also recommended by the preliminary results from by study of the Studnt PS (Rychener, 1975), discussed above in Section B.2. As an additional challenge to such an attempt, Davis and King, in a survey of the uses and characteristics of PSs (1975), say that a number of domains are inappropriate for PSs: domains involving a unified theory, as opposed to being a loose collection of diverse, independent fragments; domains that require complex kinds of control and coordination, as opposed to loosely organized ones; and domains with predetermined uses for knowledge, as opposed to having facts statable in application-free form. They don't say using PSs in such ways is impossible, but just that it is likely to be very awkward and unenlightening to do so. The present approach is in opposition to that view, and at least assumes that more evidence is required before dismissing a mechanism that has other promising features.

On the whole, a number of issues raised by work that is not in the direct tradition of this thesis will be beyond its scope: the problem of representing the statistical nature of uncertainty, as in MYCIN and in Becker's model; issues of development, as in Young and Klahr and to a lesser extent, Becker; the process of acquisition to any large extent, as investigated by the Dendral research, the MYCIN research, Waterman, Siklossy, and Hedrick; the use of Ps in a goal-driven, backwards mode instead of the recognize-act mode, as in the MYCIN system and in Becker's model; and the use of significant non-PS processing in processing Ps other than for the recognize-act cycle, as in the Dendral work. The works mentioned do raise some interesting questions, other than those topics, that might be central to the argument of the thesis and therefore might be raised again later: the level of Ps, especially with respect to the observation that MYCIN Ps are used directly for explanatory purposes; and the difficulty, also discussed in connection with MYCIN, of changing program structures that are embedded in a number of rules and that might better be represented as some other kind of global structure - in general, it may be important to observe the common kinds of modifications to Ps that are done as systems are augmented.

To repeat the emphasis of the present approach as a contrast: We want to express entire systems as PSs, avoiding approaches that have a PS as just a small augmentable subsystem, subject to a number of kinds of processing. This means that a PS represents knowledge about itself as Ps, or gains such knowledge by observing effects that occur in the global Working Memory. The other closely-related PSs raise questions certain to be touched on by the work of the thesis: whether certain augmentations can be obtained without using an ordered PS; whether variations in the PS architecture will make PSs easier to program, avoiding difficulties in maintaining local control and preventing unwanted side effects; and more generally, avoiding the need for additional control structures such as subroutines.

F. The Tasks to be Implemented as Production Systems

This section gives a preview of the tasks to be used in the body of the thesis to explore the issues of AI programming using PSs. The tasks include a simple verbal learning task, a powerful and general problem solver, a restricted chess program, a program for conversing in a restricted but natural-appearing language, and a program for conversing and performing manipulations on a toy blocks model. The tasks are chosen to be representative of observed variety in past, "classical" AI programs. There are necessarily limitations in coverage, especially of important recent work, due to limitations in available effort. Without exploring the AI field exhaustively, these programs exercise the capabilities of PSs as an expressive medium and provide a sufficiently broad basis for extrapolation to other task areas. At the end of the section, methodologies for using data gathered from the implementations will be discussed.

EPAM. The EPAM task involves having a program learn nonsense-syllable pairs under restrictions on the amount of material that can be acquired for long-term storage during the presentation of each pair (Feigenbaum, 1963). The restrictions are based on observed human behavior, but beyond that have useful properties for our exploration: information that has already been acquired must be known in some way and consistently augmented; and tests must be made in order to determine what minimum of information must be stored in order to maintain progress while staying within the restrictions. For the PS version, the information learned is represented as Ps, so that this is an example of a PS augmenting itself. It also indicates a position that can be taken with respect to using Ps to store objects more general than simple syllable pairs, namely the encoding of objects according to their distinguishing properties, in a discrimination network.

GPS. The GPS (General Problem Solver) program (Newell and Simon, 1963) uses the powerful means-ends analysis method of heuristic search to solve a variety of logic puzzles and symbolic manipulation problems. In addition to getting PSs to do heuristic search, with its attendant problems of expressing goals, maintaining goal-tree structure, and evaluating progress, GPS includes a powerful matching operation, which is able to express the differences between arbitrarily-structured objects. GPS also makes use of discrimination networks to give canonical names to various symbol structures used in the problem-solving process. Problems given to GPS are expressed independently of knowledge of the internal structure of GPS itself, so that using a PS representation for that will bring up issues of coordination and communication.

Chess endgames involving only kings and pawns. This task includes heuristic search of a slightly different form from that in GPS. It tests the ability of PSs to manipulate and access effectively representations much larger than those in GPS's repertoire. The recognition of complex patterns of elements and the description of various aspects of the board representation are also required. As Berliner (1973) has pointed out, considerable advance over past techniques of chess problem-solving will be necessary in order to attain a computer program of a Master level, to which the use of PSs may contribute.

Natural language processing. This task involves accepting descriptions, in a restricted subset of natural language, of a toy blocks scene. From the descriptions, an internal model is constructed which is used to disambiguate further inputs and to produce answers to queries. The PS program significantly extends the capabilities of its direct ancestor, the MILISY (mini-linguistic system) program of Moran (1972). The PS implementation is a fairly direct translator from external strings of words to an internal semantic representation, without recourse to conventional phrase-structure parsing or to generation of alternatives among which a search is done for the best interpretation.

Toy blocks manipulations. This task is based directly on Winograd's (1972) program for solving problems in connection with simple rearrangements of objects in a toy blocks scene. Subproblems include finding space to put unneeded objects, building stacks of objects, packing objects compactly into a space, and removing obstructions. The language capabilities of Winograd's system are only partially covered in this task, given the focus on the blocks manipulations. Instead, use is made of the language system just discussed. The PS implementation illustrates simple mechanisms for a backtracking search strategy, and easily encodes a number of task-specific heuristics, some of which can be adjusted to avoid backtracking to a large extent.

Now we turn to issues of methodology. The process of constructing these PSs and the PSs themselves provide a variety of data to be analyzed. There are some directly observable characteristics: space and time efficiency; programming time, as an indicator of productivity when using PSs; conciseness and directness of expression (obtained by noting any content of Ps that is not task-oriented so much as PS-control-oriented); features of the representation of data and procedures; aspects of how control is achieved; apparent failings or limitations of PSs; promising features of the representation as PSs with respect to extensions beyond the initial task; features that point to significant advances that might result from architectural changes within the general PS framework; features of organization of Ps, e.g., into modules; and features of the changes made to an initial design or model of the program in order to form it into a reasonable PS.

One thing to do with such traits is to use them in comparisons to other implementations. The EPAM task provides a comparison to another PS implementation, which uses a distinctly different PS architecture. The chess endgame task is currently being carried out directly in Lisp, providing a basis for a number of comparisons. Comparisons to the Planner approach to problem-solving are provided by the blocks manipulation task. Wherever the PS approach in general differs widely from the approach taken by the predecessor, close comparison is not possible or meaningful, but something should be gained with regard to general advantages or disadvantages of PSs

A couple of other methodolcgical devices prove useful. Taxonomies of the features observed, e.g. of methods of control in PSs, provide a general means towards comparisons to broad classes of other language proposals. That is, a taxonomy could be applied to other control structures in order to expose relative strengths and weaknesses. Where control devices differ sharply for two control structures, taxonomies show the kinds of issues that each is most suited to grapple with effectively, and also the kinds of issues that are likely to arise as obstacles to encoding. In addition to taxonomies that arise from observed characteristics, there are developed measures of such traits as modularity, laid out in Section B.1 as critical for building understanding systems. Such measures give support for properties of PSs such as explicitness and independence of individual Ps.

G. The Production System Design Space and Psnlst

This section gives a rough sketch of the range of design that is possible within the definition of PS architecture. Some design alternatives are suggested and implemented in Newell's PSG (Newell and McDermott, 1975). The features of Psnlst, "PS analyst", which is the architecture used for the present investigation, are introduced and placed in perspective. (Chapter II is devoted entirely to details of the Psnlst language.) It will become evident as a result of the discussion that PSs offer a considerable degree of freedom in design, and a discussion of the advantages and disadvantages of this is included. The issues of how representative the Psnlst design is and of how the design is oriented to the goals of analysis of AI programs are also discussed.

The four main components of a PS architecture are its P Memory, its Working Memory, its Recognize-Act Cycle, and its Conflict Resolution Principles. Since each P is restricted to a "condition => action" form, the primary attributes of P Memory have to do with how the Ps are related to each other (forms of condition and action elements are discussed with Working Memory, below). PSs have been used with a variety of P Memory structurings: Ps in a single linearly-ordered list; Ps subdivided into small subroutines, perhaps in hierarchies, only one of which is active at any time; and Ps in a single unstructured set. Ps in Psnlst are considered to be one unstructured set, largely in order to avoid the problem of structural context, that is, conditions in Ps implicit due to their place in some larger organizing structure rather than fully explicit. Because Psnlst's conditions are thus required to be explicit, language constructs are added to allow a P to express conditions such as whether patterns amounting to entire LHSs of other Ps are satisfied or not. If any relations do hold between Ps, they are in this way guaranteed to be explicit. Explicitness is an advantage also with respect to readability, or determination of knowledge content, although in practice it might become cumbersome to have to specify everything in this way. One side effect of coding a number of AI programs in PSs will be to decide such questions. Another advantage of keeping the P set unordered is to allow the program that interprets or otherwise executes a PS program to apply program-specific heuristics to allow it to achieve the recognition faster. For instance, certain specific P-firing sequences might be recognized as common. Condition-testing could be reordered to take advantage of that without being restricted by some structural ordering on the Ps. Finally, it seems implausible psychologically, based on the speed of human recognition and on estimates of memory size in the millions of Ps, that there is any complete linear ordering on Ps (long-term memory), although other kinds of structuring cannot be ruled out.

Forms of Working Memory range from linearly ordered to partially ordered to unordered, as regards inter-element relations; from ordered to partially ordered to unordered within elements; from flat single-level lists or sets to arbitrarily nested structures; and from pure pattern constants and variables to evocation of arbitrary functions in order to evaluate a particular pattern match. Most of the features of Psnlst's Working Memory are justified by considerations of efficiency and simplicity. The Working Memory itself is considered to be an unordered set of items, each of which has one of a set of distinguished tokens called predicates as its first element, with an ordered, uniform-sized list of arguments following the predicate name. Lack of order on the set and the presence of a predicate are considered to be efficiency and programming advantages. Working Memory elements also have status with respect to the event history of execution of the PS, as will be discussed with conflict resolution below. Working Memory is allowed to grow, in number of elements, indefinitely, a sharp contrast to psychological models, which place severe limits on size because of the correspondence with human short-term memory. (Psychologists place the limits anywhere from around 7 up to several dozens of elements; some of the limits are overcome by allowing elements to be deeply structured rather than spreading out information as a number of elements.) Element arguments may be structured lists, but they are treated simply as atomic tokens in the recognition process.

Condition and action elements tend to vary in much the same way as memory elements, since they are constrained in operating on those elements. Condition elements of Ps in Psnlst are simple abstractions of memory elements, with a required predicate constant at the head of the condition element, followed by a list of variable arguments corresponding to constant tokens in actual memory items. (Not allowing a variable at the predicate location is aimed at efficiency, and it may give some insight into the practical limitations of first-order systems (cf. first-order predicate logic).) Once a variable is bound by matching it with the corresponding token from a memory item, arbitrary Lisp predicates can be applied to test its value, ranging from equality to a constant, which is very common, to testing complex numerical relations between variables and beyond. This evaluation and testing mechanism is quite contrary to any psychological operations - humans are not considered to have such power at that low level, but must carry out arithmetic by more deliberate means - and is even controversial among pure AI researchers. Psnlst has it because it is readily available from the underlying machine structure, and I feel full advantage should be taken of it, since in other ways, such as parallelism of recognition, current computer architectures place constraints on PS efficiency and power. Not allowing nested list structures to have an effect on the match of conditions to memory elements is included for possible efficiency reasons and to make all of the essential elements more explicit, forcing their occurrence at the top level of lists in LHSs.

Action elements, although constrained like conditions by the form of Working Memory elements, do have a few added aspects, including: commands to stop the recognize-act cycle; operators to add to and modify P Memory; and operators to act on, and receive inputs from, the external (user) environment. Action elements of Ps in Psnlst are similar in form to condition elements, specifying simple additions and deletions of elements to and from Working Memory. The only exception to this is the set of P Memory modifying commands, which are expressed in form similar to other action elements, but have specific operator names in place of predicates. Other operations such as input and output are programmed as side effects: arguments to action elements can be Lisp function calls, which can be programmed to do anything externally or to compute any function on values of variables bound during the match to condition elements - access to other Working Memory elements is not provided for. Such functions must return values which are then incorporated into Working Memory elements. Psnlst doesn't enforce restrictions on action functions (because of the obvious difficulties involved as soon as anything at all is permitted), but in practice, anything beyond simple arithmetic, simple list processing, and input-output are considered to violate the spirit of PS programming. Keeping the actions basically in the same form as conditions, or at least in a relatively simple form, might eventually lead to the possibility of using the Ps in backward or "action-driven" mode, in which a chain of P firings is sought that will achieve some memory state (cf. Buchanan, 1974, or Davis et al., 1975).

The basic recognize-act component of PS architectures has been subject to the least amount of variation historically. Underlying serial hardware has predisposed systems to consist of a discrete act of recognition followed by conflict resolution followed by a sequence of serial actions, completing one cycle. Conceivably a single system could encompass a number of such cycles executing in parallel, with the same P Memory and Working Memory, either synchronized (e.g. all recognitions starting at the same time and delaying the start of the next cycle until all actions have a chance to execute), totally asynchronous, or some mixture. Considerable variation, however, does occur within the conflict resolution part of the cycle.

Conflict resolution must generally decide among a number of Ps whose conditions have been recognized as being satisfied, usually narrowing the set down to a unique choice of a P whose actions are to be executed. There are a number of system characteristics on which to base the process: the structure of Working Memory (e.g. which bindings use elements closer to the front of a linearly-ordered memory), the structure of P Memory (e.g. where the satisfied Ps stand in relation to each other in a linear ordering), the specific kind of bindings that take part in the competing recognitions (e.g., whether one is a special case of another), the history of the system (e.g. the recency of addition of Working Memory elements or Ps), the nature of the actions to be performed on the basis of the bindings (e.g. those that are indicated by the majority of bindings), random or arbitrary selection, and conceivably a number of other variations. Several principles can be combined, applied in sequence until the conflict set is narrowed down. The system can be more or less stringent on how many sequences of actions, associated with bindings found by the recognition, are allowed to be executed after the conflict resolution has been applied: uniqueness may be desired, multiple bindings to the same P may be allowed, or multiple bindings to a number of Ps may be allowed. Considerations of efficiency of implementation and of psychological plausibility are factors that influence the ultimate design of the process. Psnlst makes use primarily of the history of execution of the system, allowing those Ps to fire first that make use of the most recent Working Memory elements. Once a P has fired using some particular set of elements, it cannot fire again using the same ones (and performing the same actions), unless in the meantime one of the elements has been re-asserted into the Working Memory, effectively making it recent again. This concept is implemented using a stack, so that recent elements temporarily passed over in making a selection are pushed down on the stack, but eventually are allowed to rise back to the top, becoming candidates for selection again. Loosely speaking, elements that are most recent can be considered as events, making the system event-oriented, and giving it a focus of attention on recent events, and at the same time making it rather compulsive in exploring the consequences of all events, even when there have been numerous distracting events that have pushed them out of the immediate focus.

Using the history in this way to resolve conflicts does not determine a unique P in every situation, so Psnlst simply arbitrarily chooses one over the others. Some efficiency is gained by not even fully bringing these others into consideration: the first successful recognition found, subject to an ordering that ensures that it will be within the proper recency constraints, is executed without further ado. Another important feature, in terms of system behavior, is that if the successful recognition is able to come up with a number of possible bindings for a P, all of them are executed immediately, rather than choosing only one from the set; they are executed in arbitrary order.

We must conclude from the above considerations that the design space for PSs contains a large number of significant variations. It is difficult to know in advance how to make a decision on a number of the dimensions. With such a range of possibilities, it is even hard to sharply distinguish a PS from a non-PS, although given a particular example, there is likely to be a consensus among PS "experts". But on the other hand, this situation may not present a barrier to progress. PSs are easily implemented, so it's feasible to go through a number of design iterations fairly quickly. Also the flexibility means that PSs may be adaptable to a wide range of tasks.

To summarize how the design of Psnlst is acceptable for the overall aims of this thesis, it should first be pointed out that there are few strong assumptions made. For instance, keeping the P Memory unordered rather than having it strictly ordered as in some other PS architectures, is a weak assumption (in providing less mechanism to the user), and conclusions made from Psnlst will carry over to systems that add structure. That is, Psnlst PSs will produce similar behavior if executed in an architecture whose essential difference is in P Memory structure (we would expect secondary behavior changes such as in timing characteristics). Thus by keeping the design simple, conclusions may be more widely representative of PSs. Psnlst's design also makes few assumptions in order to be able to gather data on just which assumptions it should make. If it turns out that certain cumbersome constructions or patterns of condition elements are common, there will be strong justification for higher-level language features that make their expression easier. If we take as our aim to find ways of automating the encoding of knowledge as Ps, whether one form or another is less clumsy than the forms adopted for the present study diminishes in importance, to be replaced by considerations of openness and flexibility.

H. To the Reader

The overall structure of the thesis is general things in the outside chapters, I and VII, and detail in the others, II through VI. Chapter I introduces PSs, gives some history and a survey of other PS work, gives a priori features of PSs and their relation to research in understanding systems, and motivates the choice of tasks and the PS design used in the rest of the thesis. Chapter VII draws conclusions on the basis of the PSs constructed, reviews all of the issues covered in fragmentary fashion in the detailed chapters, and summarizes the strengths, weaknesses, and promising applications of PSs. Chapters I and VII should stand together as a unit apart from the rest of the thesis, in terms of general interest and in being free from dependence on material that is presented only in the detailed chapters. A thorough understanding of PSs cannot be effectively gained, though, without studying at least one of the inner chapters, III through VI, in detail. That study should include contact with the actual PS and its workings. Chapter II is a prerequisite to III through VI, since it introduces the Psnlst language and architecture. Note that each inner chapter covers a single task, except Chapter VI, which combines the two tasks dealing with toy blocks.

Each chapter has an abstract, a detailed table of contents (including figure titles and locations), an introduction, and usually a summary, so that it is feasible for the reader to get a general idea of the chapter's content quickly. In addition, most individual sections of chapters start with an overview and include a summary or have summaries at the ends of several subsections. Sections within chapters are labelled with capital letters, e.g., B, H, while subsections are Arabic numerals attached to section letters, e.g. B.3, H.1. Pages are numbered sequentially within chapters, and a section or subsection identifier is printed in the upper and lower corners of most pages. Appendices are also given capital letter identifiers, and are marked similarly to sections.

The structure of detailed chapters is superficially similar to the overall thesis structure: details sandwiched between more general introductions and conclusions. Each chapter gives: an overview of the task to be pursued, design issues with respect to the PS, an overview of the PS program structure and representation, examples of what the PS does in general terms, comparisons to other implementations, details, PS issues brought out by the implementation, task domain (PS-independent) issues, and conclusions. Those ingredients are not necessarily all present in every chapter, or in that order, but the reader should expect content along those lines and thus be able to be selective in what to read. The details are usually confined to one section, which usually contains: more detail on the overall PS structure and organization, more details on program behavior on an example test, a discussion of how tests were chosen for the program so that a full range of behavior could be illustrated, meanings of the predicates used in constructing the PS, and pointers to and explanations of notations in the appendices. The appendices contain program listings, a cross-reference of the Ps, and detailed program output. Each chapter has its own list of references to the AI literature. Chapter I has general PS references, while the other chapters have only the specific task-related references that are relevant local background.

Details of PSs are often presented at a general level through the use of very abstract Ps (VAPs) or abstract Ps (APs). These are an ad hoc notation that aims at describing the content of Ps without specificying all details of control, and especially neglecting local variable assignments. VAPs and APs are used to avoid such deficient devices as flowcharts, and at the same time manage to convey some of the PS spirit of the actual programs. (A similar usage appears in a few places in Newell and Simon, 1972.) VAPs generally are more abstract, corresponding to more of the actual PS per VAP than do APs. Details on the abstract notation are given in Chapter IV.

I. References

Anderson, R. H. and Gillogly, J. J., 1976. "Rand intelligent terminal agent (RITA): design philosophy", R-1809-ARPA. Santa Monica, CA: The Rand Corporation.

Becker, J. D., 1973. "A model for the encoding of experiential information", in Schank, R. C. and Colby, K. M., Eds., *Computer Models of Thought and Language*, pp. 396-434. San Francisco, CA: W. H. Freeman and Co..

Berliner, H. J., 1973. "Some necessary conditions for a master chess program", *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 77-85.

Bobrow, D. G., 1964. "A question-answering system for high-school algebra word problems", *Proc. of AFIPS Fall Joint Computer Conference, 1964*, pp. 591-614.

Bobrow, D. G. and Raphael, B. R., 1974. "New programming languages for artificial intelligence research", *Computing Surveys*, Vol. 6: 3, pp. 153-174.

Bobrow, D. G. and Wegbreit, B., 1973. "A model and stack implementation of multiple environments", *Comm. ACM*, Vol. 16: 10, pp. 591-603.

Brooks, R., 1975. "A model of human cognitive behavior in writing code for computer programs", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Buchanan, B. G., Feigenbaum, E. A. and Lederberg, J., 1971. "A heuristic programming study of theory formation in science", *Proc. Second International Joint Conference on Artificial Intelligence*, pp. 40-50. Also Stanford AI Memo 145, Stanford University Computer Science Department.

Buchanan, B. G. and Sridharan, N. S., 1973. "Analysis of behavior of chemical molecules: Rule formation on non-homogeneous classes of objects", *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 67-76. Also Stanford AI Memo 215, Stanford University Computer Science Department.

Buchanan, J. R., 1974. "A study in automatic programming", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Davies, D. and Julian, M., 1973. "Popler 1.5 reference manual", TPU Report No. 1. Edinburgh, UK: University of Edinburgh.

Davis, R., Buchanan, B. and Shortliffe, E., 1975. "Production rules as a representation for a knowledge-based consultation program", Report STAN-CS-75-519, Memo AIM-266. Stanford, CA: Stanford University, Computer Science Department.

Davis, R. and King, J., 1975. "An overview of production systems", Report STAN-CS-75-524, Memo AIM-271. Stanford, CA: Stanford University, Computer Science Department.

Enea, H. J. and Colby, K. M., 1973. "Idiolectic language analysis for understanding doctor-patient dialogues", *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 278-284.

Erman, L. D. and Lesser, V. R., 1975. "A multi-level organization for problem-solving using many, diverse, cooperating sources of knowledge", *Proc. Fourth International Joint Conference on Artificial Intelligence*, pp. 483-490.

Evans, A., 1964. "An ALGOL 60 compiler", in Goodman, R., Ed., *Annual Review of Automatic Programming*, Vol. 4, pp. 87-124. New York, NY: Pergamon Press.

Farley, A., 1974. "VIPS: A visual imagery and perception system; the result of a protocol analysis", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science. Ph. D. Thesis.

Feigenbaum, E. A., 1963. "The simulation of verbal learning behavior", in Feigenbaum, E. A. and Feldman, J., Eds., *Computers and Thought*, pp. 297-309. New York, NY: McGraw-Hill.

Fikes, R. and Nilsson, N., 1971. "STRIPS: A new approach to the application of theorem-proving to problem-solving", *Proc. Second International Joint Conference on Artificial Intelligence*, pp. 608-620.

Floyd, R., 1961. "A descriptive language for symbol manipulation", *J. ACM*, Vol. 8, pp. 579-584.

Galler, B. and Perlis, A., 1970. *A View of Programming Languages*, Reading, MA: Addison-Wesley. Especially chapters 1 and 2.

Griswold, R. E., Poage, J. F. and Polonsky, I. P., 1968. *The SNOBOL4 Programming Language*, Englewood Cliffs, NJ: Prentice-Hall. Second edition.

Hayes-Roth, F. and McDermott, J., 1976. "Knowledge acquisition from structural descriptions", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Hedrick, C. L., 1974. "A computer program to learn production systems using a semantic net", Pittsburgh, PA: Carnegie-Mellon University, Graduate School of Industrial Administration. A shortened form is in *Artificial Intelligence*, 7: 1, pp. 21-49, Spring, 1976.

Hewitt, C., 1969. "Planner: A language for proving theorems in robots", in Walker, D. E. and Norton, L. M., Eds., *Proc. First International Joint Conference on Artificial Intelligence*, pp. 167-301. Boston, MA: The Mitre Corp..

Hewitt, C., 1971. "Procedural embedding of knowledge in Planner", *Proc. Second International Joint Conference on Artificial Intelligence*, pp. 167-182.

Hewitt, C., 1972. "Description and theoretical analysis (using schemata) of Planner: A language for proving theorems and manipulating models in robots", TR-258. Cambridge, MA: MIT AI Lab.. Ph. D. Thesis.

Klahr, D., 1973. "A production system for counting, subitizing, and adding", in Chase, W. C., Ed., *Visual Information Processing*, pp. 527-546. New York, NY: Academic Press.

Klahr, D., 1976. "Steps toward the simulation of intellectual development", in Resnick, L. B., Ed., *The Nature of Intelligence*, pp. 99-133. Hillsdale, NJ: Lawrence Erlbaum Associates.

Markov, A. A., 1954. *The Theory of Algorithms*, US Dept. of Commerce, Office of Technical Services. Translated by J. J. Shorr-kon, from Teoriya Algorifmov, USSR Academy of Sciences, Moscow.

McCarthy, J., 1974. "Review of Sir J. Lighthill, Artificial intelligence: A general survey", *Artificial Intelligence*, 5, 3. pp. 317-322.

McDermott, D. V. and Sussman, G. J., 1972. "The CONNIVER reference manual", Memo 259. Cambridge, MA: MIT Artificial Intelligence Laboratory.

Minsky, M., 1967. *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ: Prentice-Hall. Chapter 12.

Minsky, M., 1975. "A framework for representing knowledge", in Winston, P. H., Ed., *The Psychology of Computer Vision*, pp. 277-211. New York, NY: McGraw-Hill.

Moore, J. and Newell, A., 1973. "How can Merlin understand?", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Moran, T. P., 1972. "MILISY: The mini-linguistic system", in Newell, A., Reddy, R., et. al., Eds., *CSD Artificial Intelligence Study Guide 72*, pp. 3.23-3.45. Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Moran, T. P., 1973. "The symbolic imagery hypothesis: An empirical investigation via a production system simulation of human behavior in a visualization task", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science. Ph. D. Thesis; short form is in *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 472-477.

Newell, A., 1967. "Studies in problem solving: Subject 3 on the cryptarithmetic task DONALD + GERALD = ROBERT", Pittsburgh, PA: Carnegie Institute of Technology.

Newell, A., 1972. "A theoretical exploration of mechanisms for coding the stimulus", in Melton, A. W. and Martin, E., Eds., *Coding Processes in Human Memory*, pp. 373-434. Washington, DC: Winston and Sons.

Newell, A., 1973. "Production systems: Models of control structures", in Chase, W. C., Ed., *Visual Information Processing*, pp. 463-526. New York, NY: Academic Press.

Newell, A. and McDermott, J., 1975. "PSG manual", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Newell, A. and Simon, H. A., 1963. "GPS, a program that simulates human thought", in Feigenbaum, E. A. and Feldman, J., Eds., *Computers and Thought*, pp. 279-293. New York, NY: McGraw-Hill.

Newell, A. and Simon, H. A., 1965. "An example of human chess play in the light of chess playing programs", in Wiener, N. and Schade, J. P., Eds., *Progress in Biocybernetics*, Vol. 2. pp. 19-75.

Newell, A. and Simon, H. A., 1972. *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall.

Nilsson, N. J., 1971. *Problem-Solving Methods in Artificial Intelligence*, New York, NY: McGraw-Hill.

Nilsson, N. J., 1974. "Artificial intelligence", Technical Note 89. Menlo Park, CA: Stanford Research Institute, Artificial Intelligence Center. Invited paper, IFIP Congress 74, Stockholm, Sweden.

Post, E., 1943. "Formal reductions of the general combinatorial decision problem", *American J. Mathematics*, Vol. 65, pp. 197-268.

Rulifson, J. F., Derksen, J. A. and Waldinger, R. J., 1972. "QA4: A procedural calculus for intuitive reasoning", AI Group Technical Note 73. Menlo Park, CA: Stanford Research Institute.

Rychener, M. D., 1975. "The Studnt production system: A study of encoding knowledge in production systems", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Shortliffe, 1974. "MYCIN: A rule based computer program for advising physicians regarding antimicrobial therapy selection", Ph.D. Thesis. Stanford, CA: Stanford University, Computer Science Department.

Siklossy, L., 1972. "Natural language learning by computer", in Simon, H. A. and Siklossy, L., Eds., *Representation and Meaning*, pp. 288-328. Englewood Cliffs, NJ: Prentice-Hall. also Ph. D. Thesis, Carnegie-Mellon University, 1968.

Simon, H. A., 1972. "On reasoning about actions", in Simon, H. A. and Siklossy, L., Eds., *Representation and Meaning*, pp. 414-430. Englewood Cliffs, NJ: Prentice-Hall.

Sussman, G. J. and McDermott, D. V., 1972. "From PLANNER to CONNIVER - A genetic approach", *Fall Joint Computer Conference*, pp. 1171-1179. Montvale, NJ: AFIPS Press.

Tesler, L. G., Enea, H. J. and Smith, D. C., 1973. "The Lisp70 pattern matching system", *Proc. Third International Joint Conference on Artificial Intelligence*, pp. 671-676.

Waterman, D. A., 1970. "Generalization learning techniques for automating the learning of heuristics", *AI*, Vol. 1, pp. 121-170.

Waterman, D. A., 1974. "Adaptive production systems", Complex Information Processing Working Paper 285. Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology. Also in *Proc. Fourth International Joint Conference on Artificial Intelligence*, pp. 296-303.

Waterman, D. A., 1975. "Serial pattern acquisition: A production system approach", Complex Information Processing Working Paper 286. Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology.

Winograd, T., 1972. *Understanding Natural Language*, New York, NY: Academic Press.

Yngve, V., 1962. "COMIT as an information retrieval language", *Comm. ACM*, Vol. 5, pp. 19-28.

Young, R. W., 1973. "Children's seriation behavior: A production-system analysis", Complex Information Processing No. 245. Pittsburgh, PA: Department of Psychology. Also available from Department of Computer Science.

Chapter II

Introduction to PsnIst

Abstract. PsnIst is a production system architecture designed for building substantial artificial intelligence systems. This chapter starts by giving an introduction to PsnIst which requires no previous experience with production systems. There is discussion of the recognize-act cycle, of the syntax of productions, and of special features. A short production system for a version of the Monkey and Bananas problem is given as an example.

Table of Contents

For Chapter II

A. Introduction

A production system (abbreviated PS) is a program consisting of a set of productions (abbreviated Ps, singular P) of the form,

conjunctive condition on Working Memory => sequence of actions

where the Working Memory is a symbolic model of a situation, and where the actions consist of additions and deletions of Working Memory elements. The action or behavior of the program results as the rules in some subset of the PS operate successively on some initial memory configuration to produce a sequence of intermediate memory states and a final state in which no conditions in the PS are satisfied. Each step in such a behavior sequence consists of a recognition of the satisfaction of some P's condition followed by execution of its actions, giving rise to the term recognize-act cycle. Psnlst, pronounced "PS analyst", is a PS architecture in which: a PS is expressed as an unordered set of Ps; the Working Memory is an unordered set of unstructured lists of symbols; and the recognize-act cycle is oriented to viewing changes to the Working Memory as attention-focusing events. Each item of Working Memory consists of an element of a set of distinguished constants called predicates, followed by an ordered list of arguments, which are usually tokens for objects. By analogy with logic terminology, a Working Memory element is referred to as an instance of its predicate, or simply as an instance.

An example of a Psnlst P is

HUNGRY(M) & ISMONKEY(M) & ISBANANAS(B) & LOC(B,X,Y,H)
=> GOTO(M,X,Y) & REACHFOR(M,B)

The portion before the "=>" is the condition or left-hand-side (LHS), the portion after, the action or right-hand-side (RHS). The identifiers M, B, X, Y, and H are variables that take on tokens as values during the match to the condition; the other identifiers are predicates. The LHS is satisfied in a model in which there is a token, say MNK-1, for which the predicates "HUNGRY" and "ISMONKEY" are satisfied, i.e., we have (ISMONKEY MNK-1) and (HUNGRY MNK-1), and in which there are tokens, say BAN-1, I-1, J-1, and K-1, such that (ISBANANAS BAN-1) and (LOC BAN-1 I-1 J-1 K-1) are true (argument positions 2, 3, and 4 of LOC are values along X-Y-Z co-ordinate axes). After recognition of the condition, the model is changed by the addition of (GOTO MNK-1 I-1 J-1) and (REACHFOR MNK-1 BAN-1). This P encodes the rule that a hungry monkey in the vicinity of some bananas tries to go to where the bananas are and tries to get its hands on them. (This model of monkey and bananas is simplified for clarity of exposition.)

Notice that the result of the application, or "firing", of this rule does not remove the condition which led to its application. This kind of infinite loop will not occur, because the Psnlst architecture makes a distinction between new data, i.e., changes to the model, and old data, that part of the model for which rule applications have already been tried. In our example model, the (ISMONKEY MNK-1) and (ISBANANAS BAN-1) are not likely to be new data, whereas (HUNGRY MNK-1) is a part of the model that is likely to change, causing examination of the above P and possibly others.

Section B of this chapter discusses in detail the processing assumptions imposed by the Psnlst architecture. The first few paragraphs should be sufficient to convey the central ideas, for a cursory reading. Section C goes through in detail an example of a PS and its execution. Section D gives a semi-formal description of Psnlst syntax, and gives semantics of special system features. The reader may want to refer to Section D while reading Section C, and vice versa.

B. The Recognize-Act Cycle

Psnlst is an event-oriented system: it starts with events from the "external world" and continues to act on the basis of internal events until no new events are evoked. Events are compulsively stacked up so that attention is brought to bear on each one, if not immediately, then at least eventually. Focusing on events serves two functions: it prevents repetitious looping in many cases and it resolves conflicts between LHSs that are simultaneously true but that do not respond to the same events. Other conflicts are resolved arbitrarily, and are taken to be either programming errors, where one of the conflicting Ps doesn't have specific enough conditions, or "don't-care" situations, where it ultimately is not supposed to matter whether one is selected before the others. This section describes the recognize-act cycle, in which a single recognition (match to an LHS) is followed by a sequence of actions (changes specified by the corresponding RHS), and whose repeated execution captures the intuitive notions just discussed.

Initially, :SMPX (stack memory for production examinations) is empty, and the system prompts the user for starting events, which are either additions or deletions of instances in Working Memory. The Ps associated with those changes are placed in :SMPX, and the basic cycle starts:

1. Try to match the LHS of the P on the top of :SMPX to instances in the Working Memory; remove that entry from :SMPX.
2. If the match fails, do nothing (i.e., skip this step), otherwise, change the Working Memory by making the insertions and deletions specified by the RHS of the P, using assignments to LHS variables made by the match. For each insertion or deletion, add associated Ps to :SMPX; this association is determined by the possibility of usage of the change in forming a match to the LHS of the P.
3. If there is anything in :SMPX, go to 1. and repeat the cycle, else prompt the user for more instances; if the user types NIL, exit the control cycle, else load up :SMPX as before and repeat the cycle starting at 1.

The preceding description outlines the basic operation of Psnlst, but leaves out several details. In order to elaborate, the following introduces some terminology and sketches briefly the necessary syntax. An LHS or RHS of a P is a conjunction, the sequence of conjuncts being separated by "&". Each conjunct consists of a predicate name and a sequence of arguments, except that in LHSs there is also a special construction consisting of a negated conjunction (details later). Except for special system predicates for evaluating Lisp predicates, conjuncts in LHSs have as arguments variables that take on Working Memory constants (tokens) as values during the match process. In RHSs, conjuncts specify how changes (additions of new instances or deletions of old ones) are to be made to the Working Memory, and arguments can be variables, quoted constants, or Lisp expressions. Conjuncts may be preceded by "NOT", which means "absence of" for LHS forms (used in the match), and "delete" for RHS forms. Conjuncts preceded by "NOT" are referred to as negative conjuncts, while others are referred to as positive.

The match performed in step 1 above is not exhaustive relative to Working Memory content, but rather is keyed to specific changes in the Working Memory. The changes used in any particular match are obtained from an :SMPX entry which associates the name of the P to be matched with a list of all the changes relevant to it that have occurred since the previous match was done on it. Those changes that are still true with respect to the current Working Memory are used to make a set of initial assignments to subsets of LHS variables. An initial assignment is made for every positive conjunct in the LHS that has the same predicate as a newly-added instance; a newly-deleted instance causes assignments to be made to variables in corresponding negative conjuncts. The match proceeds quite straight-forwardly, extending the initial assignments, trying to find instances in the Working Memory of the LHS predicates in such a way that all of the variables in LHS conjuncts are assigned in a mutually consistent way, analogous to the unification algorithm of resolution theorem-proving. In fact, there may be many such assignments that can be made, and the match returns a list of them; they are all used in the processing of step 2 for making Working Memory changes, but the order of use is indeterminate (it depends on the way the match searches the Working Memory and on the way results are formed and returned). Another important property of the match is that there is only matching at the top list level of Working Memory instances. Any complexity of instances below this top level is invisible to the match, a structure being treated simply all in one piece.

Several details of the process in step 2 above are important. The content of entries in :SMPX has already been described: a P name and a list of newly-added or -deleted instances. The RHS of a P that fires is converted to a list of changes to the Working Memory, by making variable assignments specified by the LHS match and by evaluating any Lisp expressions that are arguments. Each of these changes has associated with it a list of Ps which may be relevant to the change. Each of the Ps in the list is made into an :SMPX entry by forming a list of the name and the change. The :SMPX entries are stacked in :SMPX in such a way that the top of the stack has entries associated with the left-most change specified in the RHS, and the rest are below it in left-right order. There is one qualification to that: when an entry is stacked, if another :SMPX entry exists for the P part of the entry, the two entries are merged, the old one disappears, and the new entry on top now contains the P name and the combined list of changes from the new and old entries. So, the approximate order of :SMPX entries is determined by the left-right order of conjuncts in the RHS.

The sole use of the order in which changes occur does not determine a unique top entry in :SMPX for the simple reason that many Ps can be associated with each change. If more than one P in such an associated group should actually have satisfied LHSs, there is a conflict. As mentioned above, such conflicts are considered programming errors or "don't-care", but nevertheless Psnlst attempts to arbitrate conflicts heuristically (and without actually doing the extra computation necessary to determine the conflict, in the usual running mode). The heuristic to be described now may be seen as taking into account the relative recency of events other than the one that is common to the entries near the top of :SMPX, although for the user it is effectively indeterminate. Every time the list of Ps associated with a change is accessed it is re-ordered by a one-pass sorting algorithm that simply moves some of the Ps to the end of the list. How they are moved is based on a heuristic value associated with each P (it is the PVAL property of the P). This value is incremented every time the P is used to form an :SMPX entry, and is decremented or reduced every time a match is performed on the P. Thus, it is related to the number of changes associated with the P in its :SMPX entry. It is not a strict relation because the

B. The Recognize-Act Cycle

Psnlst is an event-oriented system: it starts with events from the "external world" and continues to act on the basis of internal events until no new events are evoked. Events are compulsively stacked up so that attention is brought to bear on each one, if not immediately, then at least eventually. Focusing on events serves two functions: it prevents repetitious looping in many cases and it resolves conflicts between LHSs that are simultaneously true but that do not respond to the same events. Other conflicts are resolved arbitrarily, and are taken to be either programming errors, where one of the conflicting Ps doesn't have specific enough conditions, or "don't-care" situations, where it ultimately is not supposed to matter whether one is selected before the others. This section describes the recognize-act cycle, in which a single recognition (match to an LHS) is followed by a sequence of actions (changes specified by the corresponding RHS), and whose repeated execution captures the intuitive notions just discussed.

Initially, :SMPX (stack memory for production examinations) is empty, and the system prompts the user for starting events, which are either additions or deletions of instances in Working Memory. The Ps associated with those changes are placed in :SMPX, and the basic cycle starts:

1. Try to match the LHS of the P on the top of :SMPX to instances in the Working Memory; remove that entry from :SMPX.
2. If the match fails, do nothing (i.e., skip this step), otherwise, change the Working Memory by making the insertions and deletions specified by the RHS of the P, using assignments to LHS variables made by the match. For each insertion or deletion, add associated Ps to :SMPX; this association is determined by the possibility of usage of the change in forming a match to the LHS of the P.
3. If there is anything in :SMPX, go to 1. and repeat the cycle, else prompt the user for more instances; if the user types NIL, exit the control cycle, else load up :SMPX as before and repeat the cycle starting at 1.

The preceding description outlines the basic operation of Psnlst, but leaves out several details. In order to elaborate, the following introduces some terminology and sketches briefly the necessary syntax. An LHS or RHS of a P is a conjunction, the sequence of conjuncts being separated by "&". Each conjunct consists of a predicate name and a sequence of arguments, except that in LHSs there is also a special construction consisting of a negated conjunction (details later). Except for special system predicates for evaluating Lisp predicates, conjuncts in LHSs have as arguments variables that take on Working Memory constants (tokens) as values during the match process. In RHSs, conjuncts specify how changes (additions of new instances or deletions of old ones) are to be made to the Working Memory, and arguments can be variables, quoted constants, or Lisp expressions. Conjuncts may be preceded by "NOT", which means "absence of" for LHS forms (used in the match), and "delete" for RHS forms. Conjuncts preceded by "NOT" are referred to as negative conjuncts, while others are referred to as positive.

The match performed in step 1 above is not exhaustive relative to Working Memory value may not be set to 0 when there is no :SMPX entry (after a match). The values are not made use of in an exact form anyway, since the sorting procedure used on the list of Ps is (for efficiency reasons) very approximate. One positive result of this heuristic is that significantly less match effort is spent finding the next matching P than is the case for random re-ordering of the P list (the "semi-sort" used is not significantly different in this regard from a strict sort on the PVAL value). An incidental effect of the re-ordering is that the ordering of the P lists quickly loses its relation to the order of Ps in the static program listing.

How the heuristic ordering works out in an actual example can be seen in the first P firing given in Section C.1. The reader may examine that and the contents of Section C.4 and Section C.5. In Section C.5, the part of the cross-reference that is used in the :SMPX processing is labelled "LHSUSES"; how that is computed should be evident from the form of the Ps in Section C.4.

The following summarizes the full detail of the Psnlst control cycle.

1. Match step
   a. Remove the top entry of :SMPX.
   b. For all of the changes noted by that entry that are still present in the Working Memory, perform a match on the entry's P.
      i. Form a set of initial assignments for each of the changes.
      ii. Try to extend each of the initial assignments, using any instances from Working Memory.
      iii. If the extension attempt succeeds, add the assignment to the list of results, if it's not already there.
   c. Reduce the PVAL value for the entry's P.
2. Action step. If in "debug" mode, check for conflicts by performing matches for the set of Ps on :SMPX that have the same first change as for the P that just matched successfully (if a conflict exists, an interactive break occurs).
   For each assignment returned by the match,
   a. Make the specified assignments, evaluate expressions, and form the list of changes.
   b. Reverse the list of changes, and process each of the changes as follows:
      i. For each P in the list associated with the change, increment the PVAL value by 1.
      ii. Semi-sort the associated list of Ps by PVAL, moving to the end of the list those with higher values. This newly-sorted list replaces the old value of the list, for use with future changes.
      iii. For each P in the list, form an :SMPX entry (adding on any changes from previous :SMPX entry, which is removed), and stack the entry on :SMPX.
      iv. Actually make the change in the Working Memory.
3. If there is anything in :SMPX, go to 1. and repeat the cycle, else prompt the user for more instances; if the user types NIL, exit the

control cycle, else load up :SMPX as before and repeat the cycle starting at 1.

C. Extended Example

This section presents a detailed run of Psnlst on a PS version of the Monkey and Bananas problem. The detail should be sufficient to provide an example of the workings of the control structure discussed in Section B, as well as presenting instances of the entities defined in the grammar of Section D. There is a full listing of the Ps in Section C.4, as well as a cross-reference of predicate uses, Section C.5.

C.1. Discussion of trace and productions

The Monkey and Bananas problem as modeled here has the monkey in a room with the bananas at an unreachable height. Three boxes are in the room, and the boxes may be stacked on top of each other to build a climbable structure for the monkey. In order to get the bananas the monkey pushes two boxes to a point under the bananas, stacks one on top of the other, climbs up, and gets the bananas. The Ps presented here represent the result of past learning on the part of the monkey: his actions are directly connected to getting the bananas, with no mistakes or searching. How these Ps get learned would be an interesting project, but is beyond the present scope. Many features of the situation that might be modeled are not, such as how the monkey knows the boxes and bananas are there (he does no looking or seeing or remembering), whether he has to avoid objects in going from one place to another and in pushing the boxes around, whether in the end it is really worthwhile for the monkey to do all this, considering the costs, risks, and benefits, and so on. Even within the specific model presented there are alternative implementations.

In what follows, the trace generated by Psnlst and the Ps of the system are intermixed with the body of text; the generated lines and the Ps are in upper case.

The first thing that must be done is to initialize the model, loading up the Working Memory with the starting situation. This is done by firing a P:

```
I1; "INIT 1" :: INIT(P)
    => EXISTS(MNK,BAN,BX1,BX2,BX3) & LOC(MNK,1,1,1) & LOC(BAN,5,5,3)
      & LOC(BX1,7,8,2) & LOC(BX2,7,8,1) & UPON(BX1,BX2) & LOC(BX3,4,6,1)
      & ISMONKEY(MNK) & ISBANANAS(BAN) & ISBOX(BX1) & ISBOX(BX2) & ISBOX(BX3)
      & HVAL(BX1,3) & HVAL(BX2,4) & HVAL(BX3,5);
```

The P is named I1, with the comment "INIT 1". The LHS is INIT(P), and the remainder is the RHS. I1 is fired by asserting an instance of INIT, as the following initial segment of the trace shows.

```
. TRACED RUN OF MONKEY FOR DOC

TOP LEVEL ASSERT (INIT PB)
INSERTING (INIT PB-1)
 EXAMINING I1 (INIT PB-1)I1/TRY
```

```
1.      I1-1        "INIT 1"
USING (INIT PB-1)
((P . PB-1))
INSERTING (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
  (LOC BX2-1 7 8 1) (UPON BX1-1 BX2-1) (LOC BX3-1 4 6 1) (ISMONKEY MNK-1)
  (ISBANANAS BAN-1) (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1) (HVAL BX1-1 3)
  (HVAL BX2-1 4) (HVAL BX3-1 5)
```

The (INIT PB) is typed by the user. It is made into the instance (INIT PB-1), inserted into the Working Memory, and processing starts. The match to the LHS of I1 is performed (as noted by I1/TRY) with respect to that instance. The match succeeds (the two lines starting at !), making use of (INIT PB-1), assigning the variable P to the object PB-1 (the two lines starting at USING). The result is the list of instances starting at INSERTING. The EXISTS in the RHS of I1 causes creation of the objects MNK-1, BAN-1, BX1-1, BX2-1, and BX3-1, which are then used to construct the instances shown, after the variables of the EXISTS are assigned them as values. The predicate LOC gives the three-coordinate location (X-axis, Y-axis, Z-axis or height). UPON indicates that two of the boxes are stacked up already, with BX1-1 on top of BX2-1. ISMONKEY, ISBOX, and ISBANANAS give the classes of the objects. Finally, HVAL is a pre-specified heuristic value that determines the order in which the boxes are chosen by the monkey (details later). (The use of this HVAL allows us to ignore how the monkey makes the choices.)

```
(       :SMPX
[ C1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (UPON BX1-1 BX2-1) (LOC BX3-1 4 6 1) (ISMONKEY MNK-1) (ISBOX BX1-1)
  (ISBOX BX2-1) (ISBOX BX3-1) ]
[ C2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) (ISMONKEY MNK-1) (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1)
  ]
[ N1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) (ISBANANAS BAN-1) (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1)
  (HVAL BX1-1 3) (HVAL BX2-1 4) (HVAL BX3-1 5) ]
[ P1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1) ]
[ H1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) (ISMONKEY MNK-1) (ISBANANAS BAN-1) ]
[ P3 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (UPON BX1-1 BX2-1) (LOC BX3-1 4 6 1) ]
[ P2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) ]
[ R2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) ]
[ R1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) ]
[ G2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) ]
[ G1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2) (LOC BX2-1 7 8 1)
  (LOC BX3-1 4 6 1) ] )
```

:SMPX now consists of the set of entries shown, which happens to include an entry for every P except I1. Each entry is in square brackets [], and it is evident that the order of entries very roughly corresponds to the number of changes relevant to the P of the entry, for example, P C1 is a candidate for further action with respect to 10 changes, C2, 9 changes, N1, 12, P1, 8, H1, 7, and so on in non-increasing order. How these relevancies are determined should not be clear, because the Ps have not been presented, but from this we can at least see some of the indeterminacy.

The system then goes through the process of testing each of the Ps in :SMPX for possible matches, and finds that none is ready to fire.

```
EXAMINING C1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (UPON BX1-1 BX2-1) (LOC BX3-1 4 6 1) (ISMONKEY MNK-1)
 (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1)C1
EXAMINING C2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1) (ISMONKEY MNK-1) (ISBOX BX1-1)
 (ISBOX BX2-1) (ISBOX BX3-1)C2
EXAMINING N1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1) (ISBANANAS BAN-1) (ISBOX BX1-1)
 (ISBOX BX2-1) (ISBOX BX3-1) (HVAL BX1-1 3) (HVAL BX2-1 4) (HVAL BX3-1 5)N1
EXAMINING P1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1) (ISBOX BX1-1) (ISBOX BX2-1) (ISBOX BX3-1)
 P1
EXAMINING H1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1) (ISMONKEY MNK-1) (ISBANANAS BAN-1)H1
EXAMINING P3 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (UPON BX1-1 BX2-1) (LOC BX3-1 4 6 1)P3
EXAMINING P2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1)P2
EXAMINING R2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1)R2
EXAMINING R1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1)R1
EXAMINING G2 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1)G2
EXAMINING G1 (LOC MNK-1 1 1 1) (LOC BAN-1 5 5 3) (LOC BX1-1 7 8 2)
 (LOC BX2-1 7 8 1) (LOC BX3-1 4 6 1)G1
```

The lack of any further action causes the system to display the Working Memory and go back to interactive mode, and this time the user types (HUNGRY 'MNK-1), which will start the monkey (MNK-1) moving.

```
HVAL (BX1-1 3) (BX2-1 4) (BX3-1 5)
INIT (PB-1)
ISBANANAS (BAN-1)
ISBOX (BX1-1) (BX2-1) (BX3-1)
ISMONKEY (MNK-1)
LOC (BAN-1 5 5 3) (BX1-1 7 8 2) (BX2-1 7 8 1) (BX3-1 4 6 1) (MNK-1 1 1 1)
UPON (BX1-1 BX2-1)

ISMONKEY (MNK-1)
TOP LEVEL ASSERT (HUNGRY (QUOTE MNK-1))
INSERTING (HUNGRY MNK-1)
```

When the monkey is hungry, he goes to where the bananas are and reaches for them:

```
H1; "HUNGRY" :: HUNGRY(M) & ISMONKEY(M) & ISBANANAS(B) & LOC(B,X,Y,H)
   => GOTO(M,X,Y) & REACHFOR(M,B);

 EXAMINING R1 (HUNGRY MNK-1)R1
 EXAMINING H1 (HUNGRY MNK-1)H1/TRY

!
2.      H1-1        "HUNGRY"
USING (HUNGRY MNK-1) (ISMONKEY MNK-1) (ISBANANAS BAN-1) (LOC BAN-1 5 5 3)
((B . BAN-1) (H . 3) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (GOTO MNK-1 5 5) (REACHFOR MNK-1 BAN-1)
```

H1 fires making use of the instances on the USING line, assigning variables as specified on the line after the USING, and inserting the instances on the INSERTING line. The GOTO and REACHFOR instances are asserted as commands whose execution is demanded of the monkey. They can be thought of as goals, in the sense that their achievement is not immediate, but requires further processing. They are a simple sequence of commands, and sequencing is handled by the ordering in :SMPX, with REACHFOR being pushed down below the GOTO, for processing after the GOTO has been achieved or attempted.

```
(         :SMPX
[ G2 (GOTO MNK-1 5 5) ]
[ G1 (GOTO MNK-1 5 5) ]
[ R2 (REACHFOR MNK-1 BAN-1) ]
[ R1 (REACHFOR MNK-1 BAN-1) ] )
```

For the GOTO action, we use a P such as,

```
G1a; "GOTO FIRST CRACK" :: GOTO(M,X,Y) & LOC(M,X2,Y2,H)
   => LOC(M,X,Y,H) & NEGATE(ALL);
```

This says that the LOC is simply changed, and the NEGATE(ALL) erases the GOTO instance and the old LOC. However, the possibility exists that the monkey is on a box so that he must climb down before the change of location. To introduce that requires that G1a is split into two Ps.

```
G1; "GOTO OK" :: GOTO(M,X,Y) & LOC(M,X2,Y2,H) & SATISFIES(H,H EQ 1)
   => LOC(M,X,Y,H) & NEGATE(ALL);

G2; "GOTO CLIMB" :: GOTO(M,X,Y) & LOC(M,X2,Y2,H) & SATISFIES(H,H ?-GREAT 1)
   => CLIMBDOWN(M) & GOTO(M,X,Y);
```

If the height is 1, meaning on the floor, the monkey goes immediately, as stated in G1. Otherwise, as G2 specifies, a CLIMBDOWN is required, followed by a repetition of the GOTO command. The repetition is necessary to add the GOTO change to :SMPX again, since it may have been removed in the processing, if G1 was looked at before G2. P splitting as just illustrated is one of the most common operations in the evolution of a PS. G2 will not fire at this point because the monkey is initially on the floor, but it will later.

```
 EXAMINING G2 (GOTO MNK-1 5 5)G2/TRY
 EXAMINING G1 (GOTO MNK-1 5 5)G1/TRY

!
```

```
3.      G1-1        "GOTO OK"
USING (GOTO MNK-1 5 5) (LOC MNK-1 1 1 1)
((H . 1) (M . MNK-1) (X . 5) (X2 . 1) (Y . 5) (Y2 . 1))
INSERTING (LOC MNK-1 5 5 1) (NOT (GOTO MNK-1 5 5)) (NOT (LOC MNK-1 1 1 1))
(          SMPX
[ R2 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) ]
[ P3 (LOC MNK-1 5 5 1) ]
[ P1 (LOC MNK-1 5 5 1) ]
[ N1 (LOC MNK-1 5 5 1) ]
[ C2 (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 1) ]
[ H1 (LOC MNK-1 5 5 1) ]
[ P2 (LOC MNK-1 5 5 1) ]
[ R1 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) ]
[ G2 (LOC MNK-1 5 5 1) ]
[ G1 (LOC MNK-1 5 5 1) ] )
 EXAMINING R2 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1)R2/TRY
```

So, now we come to the REACHFOR. The monkey is at height 1 (on the floor), reaching for the bananas at height 3, so he cannot reach them, by the assumption that he must be at the same height as the bananas to do that. (Notice that he does make the trip to the bananas, not realizing before doing it that he won't be able to reach; this is just a feature of the monkey's program for getting bananas.)

```
R2: "REACH -" :: REACHFOR(M,B) & LOC(M,X,Y,H) & LOC(B,X,Y,H2) & SATISFIES2(H,H2,H ?-LESS H2)
     & NOT( EXISTS(HN) & CLIMBUP(M,X,Y,HN) )
    => NEEDBOX(M,X,Y,H) & NEGATE(1);
```

The SATISFIES2 is a constraint on the match that H be less than H2, which is true at present. The NOT(...) is included to grapple with a problem encountered later: we don't want the monkey to reach until he has climbed up, since the failure to do the climb first will send the monkey into another box-getting cycle. NEEDBOX is the box-getting goal, and specifies the location of the desired box.

```
!
4.      R2-1        "REACH -"
USING (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 1) (LOC BAN-1 5 5 3)
((B . BAN-1) (H . 1) (H2 . 3) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (NEEDBOX MNK-1 5 5 1) (NOT (REACHFOR MNK-1 BAN-1))
(          SMPX
[ N1 (NEEDBOX MNK-1 5 5 1) (LOC MNK-1 5 5 1) ]
[ P3 (LOC MNK-1 5 5 1) ]
[ P1 (LOC MNK-1 5 5 1) ]
[ C2 (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 1) ]
[ H1 (LOC MNK-1 5 5 1) ]
[ P2 (LOC MNK-1 5 5 1) ]
[ R1 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) ]
[ G2 (LOC MNK-1 5 5 1) ]
[ G1 (LOC MNK-1 5 5 1) ] )
 EXAMINING N1 (NEEDBOX MNK-1 5 5 1) (LOC MNK-1 5 5 1)N1/TRY
```

N1 is the P that responds to the NEEDBOX instance, choosing which box to go after according to HVAL.

```
N1; "NEEDS BOX" :: NEEDBOX(M,X,Y,HN) & ISBOX(B) & LOC(B,X2,Y2,H) & NOT( VEQ(X,X2) & VEQ(Y,Y2) )
      & HVAL(B,V) & ISBANANAS(B2) & NOT( EXISTS(B3) & UPON(B3,B) )
      & NOT( EXISTS(B3,X3,Y3,H3,V3) & HVAL(B3,V3) & NOT( VEQ(X2,X3) & VEQ(Y2,Y3) )
          & SATISFIES2(V,V3,V3 ?-GREAT V) & NOT( EXISTS(B4) & UPON(B4,B3) ) )
   => GOTO(M,X2,Y2) & PUSHTO(M,B,X2,Y2,X,Y) & CLIMBUP(M,X,Y,HN) & REACHFOR(M,B2) & NEGATE(1);
```

This says, choose a box, go to the box, push it back to the bananas, climb the box, and reach for the bananas. The box is chosen by the following criteria: it must not be where the monkey is now, it must not have another box on top of it, and must have the highest HVAL of any boxes that satisfy those constraints.

```
!
5.      N1-1        "NEEDS BOX"
USING (NEEDBOX MNK-1 5 5 1) (ISBOX BX3-1) (LOC BX3-1 4 6 1) (HVAL BX3-1 5)
  (ISBANANAS BAN-1)
((B . BX3-1) (B2 . BAN-1) (H . 1) (HN . 1) (M . MNK-1) (V . 5) (X . 5)
   (X2 . 4) (Y . 5) (Y2 . 6))
INSERTING (GOTO MNK-1 4 6) (PUSHTO MNK-1 BX3-1 4 6 5 5) (CLIMBUP MNK-1 5 5 1)
  (REACHFOR MNK-1 BAN-1) (NOT (NEEDBOX MNK-1 5 5 1))
(         :SMPX
[ G1 (GOTO MNK-1 4 6) (LOC MNK-1 5 5 1) ]
[ G2 (GOTO MNK-1 4 6) (LOC MNK-1 5 5 1) ]
[ P1 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P2 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P3 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ C2 (CLIMBUP MNK-1 5 5 1) (LOC MNK-1 5 5 1) ]
[ R2 (REACHFOR MNK-1 BAN-1) ]
[ R1 (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) ]
[ C1 (LOC MNK-1 5 5 1) ]
[ H1 (LOC MNK-1 5 5 1) ] )
 EXAMINING G1 (GOTO MNK-1 4 6) (LOC MNK-1 5 5 1)G1/TRY

!
6.      G1-2        "GOTO OK"
USING (GOTO MNK-1 4 6) (LOC MNK-1 5 5 1)
((H . 1) (M . MNK-1) (X . 4) (X2 . 5) (Y . 6) (Y2 . 5))
INSERTING (LOC MNK-1 4 6 1) (NOT (GOTO MNK-1 4 6)) (NOT (LOC MNK-1 5 5 1))
(         :SMPX
[ G2 (LOC MNK-1 4 6 1) (GOTO MNK-1 4 6) (LOC MNK-1 5 5 1) ]
[ R1 (LOC MNK-1 4 6 1) (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 1)
  (REACHFOR MNK-1 BAN-1) ]
[ P2 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ H1 (LOC MNK-1 4 6 1) (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 4 6 1) (LOC MNK-1 5 5 1) ]
[ C2 (LOC MNK-1 4 6 1) (CLIMBUP MNK-1 5 5 1) (LOC MNK-1 5 5 1) ]
[ N1 (LOC MNK-1 4 6 1) ]
[ P1 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P3 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ R2 (LOC MNK-1 4 6 1) (REACHFOR MNK-1 BAN-1) ]
[ G1 (LOC MNK-1 4 6 1) ] )
 EXAMINING G2 (LOC MNK-1 4 6 1)G2
 EXAMINING R1 (LOC MNK-1 4 6 1) (REACHFOR MNK-1 BAN-1) (REACHFOR MNK-1 BAN-1)R1
  /TRY
 EXAMINING P2 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5)P2/TRY
```

The simplest case of PUSHTO is encountered first, namely, just a change in location with no unstacking or stacking. P2 is the P for this.

```
P2; "PUSH ONLY" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H)
      & NOT( EXISTS(B2,H2) & LOC(B2,X2,Y2,H2) & ISBOX(B2) )
    => LOC(M,X2,Y2,H) & LOC(B,X2,Y2,H) & NEGATE(ALL);
```

The only requirements are that the monkey and the box both be located at the box's location, at the same height, and that no box is located at the push target. The result is that the locations are changed and the PUSHTO operator deleted.

```
!
7.      P2-1        "PUSH ONLY"
USING (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 4 6 1) (LOC BX3-1 4 6 1)
((B . BX3-1) (H . 1) (M . MNK-1) (X . 4) (X2 . 5) (Y . 6) (Y2 . 5))
INSERTING (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (NOT (PUSHTO MNK-1 BX3-1 4 6 5 5))
 (NOT (LOC MNK-1 4 6 1)) (NOT (LOC BX3-1 4 6 1))
(        :SMPX
[ H1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) (LOC MNK-1 5 5 1) ]
[ C2 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) (CLIMBUP MNK-1 5 5 1)
 (LOC MNK-1 5 5 1) ]
[ N1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) ]
[ P1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P3 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ R2 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (REACHFOR MNK-1 BAN-1) ]
[ P2 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G2 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ R1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) ] )
 EXAMINING H1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 5 5 1)H1/TRY
 EXAMINING C1 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 5 5 1)C1
 EXAMINING C2 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (CLIMBUP MNK-1 5 5 1)
 (LOC MNK-1 5 5 1)C2/TRY
```

Having pushed the box to the location of the bananas, the monkey does the CLIMBUP, which causes a change in height of the monkey, and puts him UPON the box he just pushed.

```
C2; "CLIMB UP N" :: CLIMBUP(M,X,Y,H1) & LOC(M,X,Y,H) & ISMONKEY(M)
      & LOC(B1,X,Y,H1) & ISBOX(B1)
    => LOC(M,X,Y,H1+1) & UPON(M,B1) & NEGATE(1,2);
```

```
!
8.      C2-1        "CLIMB UP N"
USING (CLIMBUP MNK-1 5 5 1) (LOC MNK-1 5 5 1) (ISMONKEY MNK-1) (LOC BX3-1 5 5 1)
 (ISBOX BX3-1)
((B1 . BX3-1) (H . 1) (H1 . 1) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) (NOT (CLIMBUP MNK-1 5 5 1))
 (NOT (LOC MNK-1 5 5 1))
(        :SMPX
[ R2 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
```

```
 (REACHFOR MNK-1 BAN-1) ]
[ P3 (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1)
 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ N1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) ]
[ C1 (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) ]
[ C2 (LOC MNK-1 5 5 2) ]
[ H1 (LOC MNK-1 5 5 2) ]
[ P2 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G2 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ R1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) ]
 )
 EXAMINING R2 (LOC MNK-1 5 5 2) (LOC BX3-1 5 5 1) (REACHFOR MNK-1 BAN-1)R2/TRY

!
9.      R2-2        "REACH -"
USING (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 2) (LOC BAN-1 5 5 3)
((B . BAN-1) (H . 2) (H2 . 3) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (NEEDBOX MNK-1 5 5 2) (NOT (REACHFOR MNK-1 BAN-1))
(        SMPX
[ N1 (NEEDBOX MNK-1 5 5 2) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1)
 (LOC MNK-1 4 6 1) ]
[ P3 (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1)
 (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) ]
[ C2 (LOC MNK-1 5 5 2) ]
[ H1 (LOC MNK-1 5 5 2) ]
[ P2 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G2 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ R1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ G1 (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) ]
 )
 EXAMINING N1 (NEEDBOX MNK-1 5 5 2) (LOC MNK-1 5 5 2) (LOC BX3-1 5 5 1)N1/TRY
```

The monkey has climbed, reached, and again failed to get the bananas, so he goes through the NEEDBOX routine again.

```
!
10.     N1-2        "NEEDS BOX"
USING (NEEDBOX MNK-1 5 5 2) (ISBOX BX1-1) (LOC BX1-1 7 8 2) (HVAL BX1-1 3)
 (ISBANANAS BAN-1)
((B . BX1-1) (B2 . BAN-1) (H . 2) (HN . 2) (M . MNK-1) (V . 3) (X . 5) (X2 . 7)
 (Y . 5) (Y2 . 8)) .
INSERTING (GOTO MNK-1 7 8) (PUSHTO MNK-1 BX1-1 7 8 5 5) (CLIMBUP MNK-1 5 5 2)
 (REACHFOR MNK-1 BAN-1) (NOT (NEEDBOX MNK-1 5 5 2))
(        SMPX
[ G1 (GOTO MNK-1 7 8) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1)
 (LOC MNK-1 4 6 1) ]
[ G2 (GOTO MNK-1 7 8) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ P3 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1)
 (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
 (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P2 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) ]
[ P1 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5)
  (LOC MNK-1 5 5 1) ]
[ C2 (CLIMBUP MNK-1 5 5 2) (LOC MNK-1 5 5 2) ]
[ R2 (REACHFOR MNK-1 BAN-1) ]
[ R1 (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) ]
[ H1 (LOC MNK-1 5 5 2) ] )
 EXAMINING G1 (GOTO MNK-1 7 8) (LOC MNK-1 5 5 2) (LOC BX3-1 5 5 1)G1/TRY
 EXAMINING G2 (GOTO MNK-1 7 8) (LOC MNK-1 5 5 2) (LOC BX3-1 5 5 1)G2/TRY

!
11.     G2-1         "GOTO CLIMB"
USING (GOTO MNK-1 7 8) (LOC MNK-1 5 5 2)
((H . 2) (M . MNK-1) (X . 7) (X2 . 5) (Y . 8) (Y2 . 5))
WARNING (MNK-1 7 8) ALREADY UNDER GOTO **
INSERTING (CLIMBDOWN MNK-1) (GOTO MNK-1 7 8)
(          SMPX
[ C1 (CLIMBDOWN MNK-1) (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) ]
[ G2 (GOTO MNK-1 7 8) ]
[ G1 (GOTO MNK-1 7 8) ]
[ P3 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1)
  (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
  (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P2 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) ]
[ P1 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1) (PUSHTO MNK-1 BX3-1 4 6 5 5)
  (LOC MNK-1 5 5 1) ]
[ C2 (CLIMBUP MNK-1 5 5 2) (LOC MNK-1 5 5 2) ]
[ R2 (REACHFOR MNK-1 BAN-1) ]
[ R1 (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 2) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) ]
[ H1 (LOC MNK-1 5 5 2) ] )
 EXAMINING C1 (CLIMBDOWN MNK-1) (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1)C1/TRY
```

This time, it is necessary to climb down before the GOTO operation.

```
C1: "CLIMB DOWN" :: CLIMBDOWN(M) & LOC(M,X,Y,H) & UPON(M,B) & ISBOX(B) & ISMONKEY(M)
   => LOC(M,X,Y,1) & NEGATE(1,2,3);

!
12.     C1-1         "CLIMB DOWN"
USING (CLIMBDOWN MNK-1) (LOC MNK-1 5 5 2) (UPON MNK-1 BX3-1) (ISBOX BX3-1)
  (ISMONKEY MNK-1)
((B . BX3-1) (H . 2) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (LOC MNK-1 5 5 1) (NOT (CLIMBDOWN MNK-1)) (NOT (LOC MNK-1 5 5 2))
  (NOT (UPON MNK-1 BX3-1))
(          SMPX
[ R1 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 2)
  (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ P2 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2)
  (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) ]
[ C2 (LOC MNK-1 5 5 1) (CLIMBUP MNK-1 5 5 2) (LOC MNK-1 5 5 2) ]
[ P1 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2)
  (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
  (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ P3 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 2)
  (UPON MNK-1 BX3-1) (LOC MNK-1 5 5 1) (LOC BX3-1 5 5 1) (LOC MNK-1 4 6 1)
  (PUSHTO MNK-1 BX3-1 4 6 5 5) (LOC MNK-1 5 5 1) ]
[ R2 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) ]
[ N1 (LOC MNK-1 5 5 1) ]
[ C1 (LOC MNK-1 5 5 1) ]
[ H1 (LOC MNK-1 5 5 1) (LOC MNK-1 5 5 2) ]
[ G2 (LOC MNK-1 5 5 1) (GOTO MNK-1 7 8) ]
[ G1 (LOC MNK-1 5 5 1) (GOTO MNK-1 7 8) ] )
 EXAMINING R1 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1)R1/TRY
 EXAMINING P2 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1)P2/TRY
 EXAMINING C2 (LOC MNK-1 5 5 1) (CLIMBUP MNK-1 5 5 2)C2/TRY
 EXAMINING P1 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) (LOC MNK-1 5 5 1)P1/TRY
 EXAMINING P3 (LOC MNK-1 5 5 1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 5 5 1)
  (LOC BX3-1 5 5 1) (LOC MNK-1 5 5 1)P3/TRY
 EXAMINING R2 (LOC MNK-1 5 5 1) (REACHFOR MNK-1 BAN-1)R2/TRY
 EXAMINING N1 (LOC MNK-1 5 5 1)N1
 EXAMINING C1 (LOC MNK-1 5 5 1)C1
 EXAMINING H1 (LOC MNK-1 5 5 1)H1/TRY
 EXAMINING G2 (LOC MNK-1 5 5 1) (GOTO MNK-1 7 8)G2/TRY
 EXAMINING G1 (LOC MNK-1 5 5 1) (GOTO MNK-1 7 8)G1/TRY

)
13.    G1-3         "GOTO OK"
USING (GOTO MNK-1 7 8) (LOC MNK-1 5 5 1)
((H . 1) (M . MNK-1) (X . 7) (X2 . 5) (Y . 8) (Y2 . 5))
INSERTING (LOC MNK-1 7 8 1) (NOT (GOTO MNK-1 7 8)) (NOT (LOC MNK-1 5 5 1))
(          SMPX
[ P3 (LOC MNK-1 7 8 1) ]
[ P1 (LOC MNK-1 7 8 1) ]
[ C2 (LOC MNK-1 7 8 1) ]
[ P2 (LOC MNK-1 7 8 1) ]
[ R1 (LOC MNK-1 7 8 1) ]
[ G1 (LOC MNK-1 7 8 1) ]
[ R2 (LOC MNK-1 7 8 1) ]
[ N1 (LOC MNK-1 7 8 1) ]
[ C1 (LOC MNK-1 7 8 1) ]
[ H1 (LOC MNK-1 7 8 1) ]
[ G2 (LOC MNK-1 7 8 1) ] )
 EXAMINING P3 (LOC MNK-1 7 8 1)P3/TRY
```

The PUSHTO in this case is on a box that is UPON another box, so that an unstack is necessary before the execution of the PUSHTO. The unstack is done by a simple change in location, without an explicit operation, since the operation for unstack would use all of the same information that appears in the LHS of P3.

```
P3; "UNSTACK BEFORE PUSH" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H2) & UPON(B,B2)
   => PUSHTO(M,B,X,Y,X2,Y2) & LOC(B,X,Y,1) & NEGATE(3,4);
```

```
!
14.    P3-1        "UNSTACK BEFORE PUSH"
USING (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 7 8 1) (LOC BX1-1 7 8 2)
 (UPON BX1-1 BX2-1)
((B . BX1-1) (B2 . BX2-1) (H . 1) (H2 . 2) (M . MNK-1) (X . 7) (X2 . 5) (Y . 8)
 (Y2 . 5))
WARNING (MNK-1 BX1-1 7 8 5 5) ALREADY UNDER PUSHTO ..
INSERTING (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) (NOT (LOC BX1-1 7 8 2))
 (NOT (UPON BX1-1 BX2-1))
(      SMPX
[ P1 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ P2 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ P3 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) ]
[ R1 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ C2 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ G1 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ R2 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ N1 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ C1 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ H1 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ G2 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ] )
 EXAMINING P1 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1)
 P1/TRY
```

A second variation in the PUSHTO is that there already exists a box at the target location, so that an immediate stack operation is performed *(implicitly)* by a change in location and the addition of the UPON predicate, as follows.

```
P1: "PUSH & STACK" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H)
      & LOC(B2,X2,Y2,H2) & ISBOX(B2) & NOT( EXISTS(B3) & UPON(B3,B2) )
    => LOC(M,X2,Y2,H) & LOC(B,X2,Y2,H2+1) & UPON(B,B2) & NEGATE(1,2,3);
```

```
!
15.    P1-1        "PUSH & STACK"
USING (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC MNK-1 7 8 1) (LOC BX1-1 7 8 1)
 (LOC BX3-1 5 5 1) (ISBOX BX3-1)
((B . BX1-1) (B2 . BX3-1) (H . 1) (H2 . 1) (M . MNK-1) (X . 7) (X2 . 5) (Y . 8)
 (Y2 . 5))
INSERTING (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (UPON BX1-1 BX3-1)
 (NOT (PUSHTO MNK-1 BX1-1 7 8 5 5)) (NOT (LOC MNK-1 7 8 1))
 (NOT (LOC BX1-1 7 8 1))
(      SMPX
[ R1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ P2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (PUSHTO MNK-1 BX1-1 7 8 5 5)
 (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ C2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ P1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) ]
[ P3 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (UPON BX1-1 BX3-1)
 (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) ]
[ G1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ R2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ N1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ C1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (UPON BX1-1 BX3-1) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ H1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ G2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
 )
```

```
EXAMINING R1 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2)R1/TRY
EXAMINING P2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2)P2
EXAMINING C2 (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2)C2/TRY

!
16.     C2-2        "CLIMB UP N"
USING (CLIMBUP MNK-1 5 5 2) (LOC MNK-1 5 5 1) (ISMONKEY MNK-1) (LOC BX1-1 5 5 2)
 (ISBOX BX1-1)
((B1 . BX1-1) (H . 1) (H1 . 2) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (LOC MNK-1 5 5 3) (UPON MNK-1 BX1-1) (NOT (CLIMBUP MNK-1 5 5 2))
 (NOT (LOC MNK-1 5 5 1))
(       SMPX
[ G2 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ H1 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ C1 (LOC MNK-1 5 5 3) (UPON MNK-1 BX1-1) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2)
 (UPON BX1-1 BX3-1) (LOC BX1-1 7 8 1) (LOC MNK-1 7 8 1) ]
[ N1 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ R2 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ G1 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) (LOC BX1-1 7 8 1)
 (LOC MNK-1 7 8 1) ]
[ P3 (LOC MNK-1 5 5 3) (UPON MNK-1 BX1-1) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2)
 (UPON BX1-1 BX3-1) (PUSHTO MNK-1 BX1-1 7 8 5 5) (LOC BX1-1 7 8 1) ]
[ P1 (LOC MNK-1 5 5 3) (LOC MNK-1 5 5 1) (LOC BX1-1 5 5 2) ]
[ C2 (LOC MNK-1 5 5 3) ]
[ R1 (LOC MNK-1 5 5 3) ]
[ P2 (LOC MNK-1 5 5 3) ] )
EXAMINING G2 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)G2
EXAMINING H1 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)H1/TRY
EXAMINING C1 (LOC MNK-1 5 5 3) (UPON MNK-1 BX1-1) (LOC BX1-1 5 5 2)
 (UPON BX1-1 BX3-1)C1
EXAMINING N1 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)N1
EXAMINING R2 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)R2/TRY
EXAMINING G1 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)G1
EXAMINING P3 (LOC MNK-1 5 5 3) (UPON MNK-1 BX1-1) (LOC BX1-1 5 5 2)
 (UPON BX1-1 BX3-1)P3
EXAMINING P1 (LOC MNK-1 5 5 3) (LOC BX1-1 5 5 2)P1
EXAMINING C2 (LOC MNK-1 5 5 3)C2
EXAMINING R1 (LOC MNK-1 5 5 3)R1/TRY
```

Finally, the monkey is at the proper height for reaching the bananas, and he does the appropriate thing.

```
R1: "REACH +" :: REACHFOR(M,B) & LOC(M,X,Y,H) & LOC(B,X,Y,H) & HUNGRY(M)
  => EATS(M,B) & NEGATE(ALL,-2);
```

```
!
17.     R1-1        "REACH +"
USING (REACHFOR MNK-1 BAN-1) (LOC MNK-1 5 5 3) (LOC BAN-1 5 5 3) (HUNGRY MNK-1)
((B . BAN-1) (H . 3) (M . MNK-1) (X . 5) (Y . 5))
INSERTING (EATS MNK-1 BAN-1) (NOT (REACHFOR MNK-1 BAN-1))
 (NOT (LOC BAN-1 5 5 3)) (NOT (HUNGRY MNK-1))
(       SMPX
[ P2 (LOC MNK-1 5 5 3) ] )
 EXAMINING P2 (LOC MNK-1 5 5 3)P2
```

The following gives the entire Working Memory contents at the end of the run, plus some run statistics.

```
EATS (MNK-1 BAN-1)
HVAL (BX1-1 3) (BX2-1 4) (BX3-1 5)
INIT (PB-1)
ISBANANAS (BAN-1)
ISBOX (BX1-1) (BX2-1) (BX3-1)
ISMONKEY (MNK-1)
LOC (BX1-1 5 5 2) (BX2-1 7 8 1) (BX3-1 5 5 1) (MNK-1 5 5 3)
UPON (BX1-1 BX3-1) (MNK-1 BX1-1)

ISMONKEY (MNK-1)
EATS (MNK-1 BAN-1)
TOP LEVEL ASSERT NIL

RUN TIME 19.5 SEC

EXAM   TRY     FIRE      WMACT    E/F      E/T      T/F
57     32      17        74       3.35     1.78     1.88
0.342  0.610   1.15      0.264    SEC AVG

46 INSERTS 28 DELETES 2 WARNINGS 6 NEW OBJECTS
MAX :SMPX LENGTH 11
CORE (FREE.FULL): (10880 . 883) USED (758 . 27)

:ACTS SETUP (MONKEY . LSP) SAVEPS (CLOSED (MONKEY . EXP)) MONC RUN SMPXEMPTY
  SMPXEMPTY

TRACE
(I1-1 H1-1 G1-1 R2-1 N1-1 G1-2 P2-1 C2-1 R2-2 N1-2 G2-1 C1-1 G1-3 P3-1 P1-1
  C2-2 R1-1)

FIRED 12 OUT OF 12 PRODS
```

The overall control of this run was achieved through the command (CMD) file MONC, loaded as a result of the DCMD declaration in the program (see Section C.4). The contents of MONC are:

```
:CYCLECMDS '((DUMP) (DUMPQ ISMONKEY EATS))
:TERMCMDS '((PERFEVAL T) (ERASE T))
:DEBUG T
:DISPDEPTH 99
```

The first line gives the :CYCLECMDS used in the run. These are executed each time the :SMPX becomes empty, and allows the user to cycle again without saying *RUN*, after executing the commands. The commands cause display of the whole Working Memory, plus the display of two predicates that are important for the run. The :TERMCMDS tell what to do when NIL is typed to the prompt for TOP LEVEL ASSERT. The results of those commands are at the end of the run above. The last two commands set the :DEBUG switch, for detection of conflicts, and the display depth for :SMPX.

C.2. Concluding comments on the example

There are several interesting features of this program, and Psnlst programs in general, that should be emphasized:

a. The Working Memory is large (compared to other PS architectures), but :SMPX provides a focus for the processing, acting as an attention mechanism.
b. The conditions of the Ps are mutually exclusive. This means that in order to add Ps, closely related conditions must be consulted. Also, when a P is split into two (cf. G1a above), both halves have longer LHSs.
c. The stack implementation of :SMPX gives a depth-first, goal-stacking organization. We saw in P G2 how explicit re-assertion of an element already in Working Memory brings it to a higher stack position, which adds flexibility.
d. Flow of control in the program is dependent on the content of the Working Memory, on the changes made to it, and on the order of those changes. Rarely does one P signal a single other P; rather, a signal is emitted to a set of Ps, and the condition of the Working Memory relative to the LHSs of the receivers determines which one makes use of the signal. This is analogous to emitting a goal and letting a variety of methods decide whether to work on it.

There is one prominent characteristic of this example that is atypical of Psnlst PSs. Every time there was a change in the LOC instances, an :SMPX entry was made for each P in the system. Ordinarily, a predicate of such universal usage is declared to be a non-fluent, to prevent this heavy loading of :SMPX (consequences of changes made to a non-fluent are not explored, and no :SMPX entries are made for such changes). The declaration was left out in order to avoid complications in the illustration of the :SMPX processing. It is also the case that LOC was a key instance for at least one match, leading to the firing of P3. But, in fact, the system works the same way with LOC as a non-fluent, because the LOC as used above causes pre-mature examination of P3, removal of its :SMPX entry, and then addition of another :SMPX entry, thus cancelling out the ill effects. The pre-mature examination resulted in the loss from :SMPX of the PUSHTO goal, and the change in location in a sense jarred the memory of the monkey to bring the PUSHTO into the match process. In general, in cases where such pre-mature examination is unavoidable, use must be made of explicit remember-goal markers, which cause a P to fire, inserting a new incarnation of the goal marker. Such cases are rare, and the :SMPX stacking regime usually suffices to hold goals in the wings until conditions are right.

C.3. A note on reading productions

In trying to determine the intent of a P, there are a few heuristics that may help. Each P has, in general, one principal idea or piece of knowledge. This is its essential action, and can be obtained by looking at the first few conjuncts in its LHS and RHS. For instance, in N1, the P that represents what happens when the monkey needs a box to reach some high place, the first three LHS conjuncts can be combined with the first two RHS conjuncts to get the principal idea of the P, that the monkey goes to the box and pushes it to a location under the place.

```
N1: "NEEDS BOX" :: NEEDBOX(M,X,Y,HN) & ISBOX(B) & LOC(B,X2,Y2,H) . . .
   => GOTO(M,X2,Y2) & PUSHTO(M,B,X2,Y2,X,Y) . . . ;
```

The other LHS conjuncts only elaborate the necessary side conditions and the remainder of the RHS gives secondary actions and peripheral updating.

How it helps to have the main conjuncts requires a more detailed explanation. Each predicate is given a meaning (see Section C.5), a proposition that relates its arguments to each other. Conjuncts with shared variables result in extending and merging the relations between arguments. In addition, the contrast between LHS and RHS enters in, namely in establishing "before" and "after" properties. For instance, in N1, the first RHS conjunct shares variables with the first and third LHS conjuncts in a way that also interacts with the LHS-RHS meaning to arrive at the "monkey goes to the box" part of the principal idea. To summarize, the main trick here is to look at both LHS and RHS simultaneously rather than attending too soon to the side conditions in the LHS.

C.4. Program listing

```
BEGIN          % PS FOR MONKEY AND BANANAS %

EXPR MONKEY(); BEGIN          DCMD(MONC);

H1; "HUNGRY" :: HUNGRY(M) & ISMONKEY(M) & ISBANANAS(B) & LOC(B,X,Y,H)
   => GOTO(M,X,Y) & REACHFOR(M,B);

G1; "GOTO OK" :: GOTO(M,X,Y) & LOC(M,X2,Y2,H) & SATISFIES(H,H EQ 1)
   => LOC(M,X,Y,H) & NEGATE(ALL);

G2; "GOTO CLIMB" :: GOTO(M,X,Y) & LOC(M,X2,Y2,H) & SATISFIES(H,H ?.GREAT 1)
   => CLIMBDOWN(M) & GOTO(M,X,Y);

C1; "CLIMB DOWN" :: CLIMBDOWN(M) & LOC(M,X,Y,H) & UPON(M,B) & ISBOX(B) & ISMONKEY(M)
   => LOC(M,X,Y,1) & NEGATE(1,2,3);

C2; "CLIMB UP N" :: CLIMBUP(M,X,Y,H1) & LOC(M,X,Y,H) & ISMONKEY(M)
      & LOC(B1,X,Y,H1) & ISBOX(B1)
   => LOC(M,X,Y,H1+1) & UPON(M,B1) & NEGATE(1,2);

R1; "REACH +" :: REACHFOR(M,B) & LOC(M,X,Y,H) & LOC(B,X,Y,H) & HUNGRY(M)
   => EATS(M,B) & NEGATE(ALL,-2);

R2; "REACH -" :: REACHFOR(M,B) & LOC(M,X,Y,H) & LOC(B,X,Y,H2) & SATISFIES2(H,H2,H ?.LESS H2)
      & NOT( EXISTS(HN) & CLIMBUP(M,X,Y,HN) )
   => NEEDBOX(M,X,Y,H) & NEGATE(1);

N1; "NEEDS BOX" :: NEEDBOX(M,X,Y,HN) & ISBOX(B) & LOC(B,X2,Y2,H) & NOT( VEQ(X,X2) & VEQ(Y,Y2) )
      & HVAL(B,V) & ISBANANAS(B2) & NOT( EXISTS(B3) & UPON(B3,B) )
      & NOT( EXISTS(B3,X3,Y3,H3,V3) & HVAL(B3,V3) & NOT( VEQ(X2,X3) & VEQ(Y2,Y3) )
            & SATISFIES2(V,V3,V3 ?.GREAT V) & NOT( EXISTS(B4) & UPON(B4,B3) ) )
   => GOTO(M,X2,Y2) & PUSHTO(M,B,X2,Y2,X,Y) & CLIMBUP(M,X,Y,HN) & REACHFOR(M,B2) & NEGATE(1);

P1; "PUSH & STACK" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H)
      & LOC(B2,X2,Y2,H2) & ISBOX(B2) & NOT( EXISTS(B3) & UPON(B3,B2) )
   => LOC(M,X2,Y2,H) & LOC(B,X2,Y2,H2+1) & UPON(B,B2) & NEGATE(1,2,3);

P2; "PUSH ONLY" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H)
      & NOT( EXISTS(B2,H2) & LOC(B2,X2,Y2,H2) & ISBOX(B2) )
   => LOC(M,X2,Y2,H) & LOC(B,X2,Y2,H) & NEGATE(ALL);

P3; "UNSTACK BEFORE PUSH" :: PUSHTO(M,B,X,Y,X2,Y2) & LOC(M,X,Y,H) & LOC(B,X,Y,H2) & UPON(B,B2)
   => PUSHTO(M,B,X,Y,X2,Y2) & LOC(B,X,Y,1) & NEGATE(3,4);

I1; "INIT 1" :: INIT(P)
   => EXISTS(MNK,BAN,BX1,BX2,BX3) & LOC(MNK,1,1,1) & LOC(BAN,5,5,3)
      & LOC(BX1,7,8,2) & LOC(BX2,7,8,1) & UPON(BX1,BX2) & LOC(BX3,4,6,1)
      & ISMONKEY(MNK) & ISBANANAS(BAN) & ISBOX(BX1) & ISBOX(BX2) & ISBOX(BX3)
      & HVAL(BX1,3) & HVAL(BX2,4) & HVAL(BX3,5);

END; END
```

C.5. Cross-reference and meanings for predicates

```
CLIMBDOWN(m) - monkey m is to climb down from some elevated object (e.g. box).
  LHSUSES C1
  RHSUSES G2 -C1
CLIMBUP(m,x,y,h) - monkey is to climb up to height h at co-ordinates <x,y>.
  LHSUSES C2
  NESTEDL R2
  RHSUSES -C2 N1
EATS(m,bn) - monkey m eats bn (in our case, bananas).
  RHSUSES R1
GOTO(m,x,y) - monkey is to go to co-ordinates <x,y>.
  LHSUSES G1 G2
  RHSUSES H1 -G1 G2 N1
HUNGRY(m) - m is hungry.
  LHSUSES H1 R1
  RHSUSES -R1
HVAL(bx,n) - bx has heuristic value n, which orders how objects are chosen.
  LHSUSES N1
  NESTEDL N1
  RHSUSES I1
INIT(p) - initialize problem; p is a dummy.
  LHSUSES I1
ISBANANAS(bn) - bn is a bunch of bananas.
  LHSUSES H1 N1
  RHSUSES I1
ISBOX(bx) - bx is a box.
  LHSUSES C1 C2 N1 P1
  NESTEDL P2
  RHSUSES I1
ISMONKEY(m) - m is a monkey.
  LHSUSES H1 C1 C2
  RHSUSES I1
LOC(o,x,y,h) - object o is located at co-ordinates <x,y>, height h.
  LHSUSES H1 G1 G2 C1 C2 R1 R2 N1 P1 P2 P3
  NESTEDL P2
  RHSUSES G1 -G1 C1 -C1 C2 -C2 -R1 P1 -P1 P2 -P2 P3 -P3 I1
NEEDBOX(m,x,y,h) - monkey needs to move a box to <x,y>, height h.
  LHSUSES N1
  RHSUSES R2 -N1
PUSHTO(m,bx,x1,y1,x2,y2) - monkey m is to push box bx from <x1,y1> to <x2,y2>.
  LHSUSES P1 P2 P3
  RHSUSES N1 -P1 -P2 P3
REACHFOR(m,bn) - monkey m is to reach for bananas bn.
  LHSUSES R1 R2
  RHSUSES H1 -R1 -R2 N1
UPON(o,bx) - o is upon box bx.
  LHSUSES C1 P3
  NESTEDL N1 P1
  RHSUSES -C1 C2 P1 -P3 I1
```

D. Grammar for Psnlst

Syntactic meta-variables are in lower case. The suffix "-x-list" refers to a list of one or more of the entities specified by the part of the variable before the first "-", separated by the delimiting character or grammar entity at the position "x"; for instance "argument-,-list" is a list of argument's, separated by ",". Alternatives in the grammar are separated by "|". ":=" separates definition from what is being defined, if the definition is a formal definition; for informal ones, "is" is used. Optional sequences are enclosed by "[" and "]". The order of definition of the grammar variables is depth-first by the line of first appearance, and within lines, left-to-right. That means, as the definitions proceed, the most recently-mentioned variables get defined next, with others before them stacked up for later definition, in a last-in first-out order except that the stack removal follows left-to-right order within definitions.

```
system          := BEGIN [ define-;-list ; ] module-;-list END.
supercomment    is a comment that may be inserted anywhere, and is
                       enclosed in %'s ; the % character may not be used singly
                       elsewhere, even in identifiers or inside pairs of "'s
define          := DEFINE oldid newid
oldid           := ident
ident           is a string of characters, where a character can be
                       a letter, a digit, :, !, or ? followed by anything;
                       the first character of an ident cannot be a digit;
                       examples: I1 :VAR?-3 Fire! X?.PROC ?*QUO ?95K
newid           := ident
module          := EXPR modulename ( ) ; [ declare-;-list ; ] prod-;-list ; END
modulename      := ident
declare         := declareword ( identpair-,-list )
declareword     := REQUIRE | NONFLUENT | DCMD | PSMACRO
identpair       := ident | ' ( ident . ident )
prod            := prodname ; [ comnt orchar ] lhs arrchar rhs
prodname        := ident
comnt           := " string "
string          is a string of characters, except " and %
orchar          is a character or DEFINE'd ident that stands for OR;
                       OR itself could be used; common practice is to use :: ;
                       the ascii character for OR is 37 (octal), which
                       prints as SOS ?8 or ↑←
lhs             := lhsconj-andchar-list
lhsconj         := lhspredarg | notchar lhspredarg | notchar ( nestedconj )
                       | psmacrocall
lhspredarg      := lhsspecial ( lspecarg-,-list ) | pred ( var-,-list )
lhsspecial      := SATISFIES | SATISFIES2 | SATISFIES3 | VNEQ | VEQ
lspecarg        is a var or exp, depending on the particular
                       special: VNEQ and VEQ take two var arguments;
                       the three SATISFIES's take one, two, and three var
                       arguments, followed by one argument which is an exp
var             := ident
exp             is an Mlisp expression or a quoted Lisp expression
```

	(see the notes following the grammar)
pred	:= ident
notchar	is NOT or ascii character 5 (may be DEFINE'd otherwise)
nestedconj	:= exists andchar lhs | lhsconj andchar lhs
exists	:= EXISTS (var-,-list)
andchar	is AND or & or ascii character 4 (may be DEFINE'd otherwise)
psmacrocall	is a call on a user-defined psmacro; it must return a value with the format of an LHS or RHS, depending on where it occurs
arrchar	is a character or DEFINE'd ident that stands for OR; OR itself could be used; a modified Mlisp (PSNPRE) is required to be able to use "=>" (which appears in the Ps in this and other documents); ascii 37 (octal) is OR, printing as SOS ?8 or ↑←
rhs	:= rhsconj-andchar-list
rhsconj	:= pred (varexp-,-list) | notchar pred (varexp-,-list) | exists | NEGATE (negargs) | rhsspecial (rspecarg-,-list) | psmacrocall
varexp	:= var | exp
negargs	:= posint-,-list | ALL | ALL , negint-,-list
posint	is a positive integer
negint	is a negative integer
rhsspecial	:= DELAYEXPND | ADDPROD | REPPROD | REPLHS | REPRHS | REPCOMNT
rspecarg	is a varexp-,-list ; the number of varexp's depends on rhsspecial : 1, 5, 4, 2, 2, 2, respectively

The following gives the meanings for declareword's, lhsconj's, lhsspecial's, rhsconj's, EXISTS, NEGATE, and rhsspecial's.

REQUIRE causes the modules whose names are arguments to be loaded automatically whenever the module containing the declaration is loaded (by LOADPS, the PS load function). Extension EXP is assumed for those files. Example: REQUIRE(STUT, STUCR) causes STUT.EXP and STUCR.EXP to be loaded.

NONFLUENT causes its arguments, which are assumed to be pred's, to become non-fluents, that is, when changes to the Working Memory are made on these pred's, no :SMPX entry is created for following up any consequences of those changes. Example: NONFLUENT(LEFTOF, WORDEQ).

DCMD causes its argument to be passed to the function CMD, whenever the containing module is loaded or set up (LOADPS or SETUP functions). CMD is the function which reads a command file. Example: DCMD('(STUDNT.CMD)).

PSMACRO declares its arguments to be PSMACRO files; they are read immediately on being declared, resulting in definition of the functions they contain; extension MAC is assumed. The format of the files should be Lisp DEFPROP format or its equivalent. Example: PSMACRO(STUDNM).

lhsconj specifies a template to be used during the match process to test for presence or absence of an instance in the Working Memory. As each lhsconj is matched, variables with value NIL are assigned corresponding values from the matching instance. The Lisp VALUE property is used for this binding. A notchar preceding a predarg specifies absence of a particular instance, that is, all variables in the var-,-list must have been used previously in the LHS and thereby bound. The notchar (nestedconj) construct can be used to specify absence of a more complex condition, allowing quantification over variables via EXISTS (see below), and otherwise allowing negation of a conjunction of templates. A match is attempted on whatever is inside the ()'s, and if that match fails, the main match continues.

SATISFIES tests the value of the Mlisp or Lisp expression which is its second argument; the match is allowed to continue if the result is non-NIL. The first argument is a variable which is to be used as an argument to the expression (it also explicitly appears in the expression to be evaluated, of course). In this as in other places, if a QUOTEd Lisp expression appears, the CADR of the QUOTE expression is evaluated. This allows a user to express expressions either in Mlisp or in Lisp. For instance, one could say SATISFIES(X, NUMBERP X) or SATISFIES(X, '(NUMBERP X)) with the same result. Another special feature of SATISFIES (and the other lhsspecials) is that preceding it by NOT causes the internal result to have the NOT around the expression, for instance NOT SATISFIES(N, N ?*LESS 7) becomes internally (SATISFIES N (NOT (*LESS N 7)))

SATISFIES2 is similar to SATISFIES, with two variables declared to be needed for the third expression argument. Example: SATISFIES2(X, Y, X ?*GREAT Y) or SATISFIES2(X, Y, '(?*GREAT X Y).

SATISFIES3 takes three variables and an expression, for instance, SATISFIES3(L, M, N, N = L + M) or SATISFIES3(L, M, N, '(EQUAL N (PLUS L M))).

VNEQ compares the values of its two variables, and allows the match to continue if they are different (not EQ). Example: VNEQ(C, D).

VEQ compares the values of its two variables, allowing the match to continue if they are the same (EQ). NOT VEQ(X, Y) is converted internally to VNEQ(X, Y). NOT VNEQ(C, D) is converted to VEQ(C, D).

EXISTS is used in nestedconj's in LHSs to declare a set of variables in the local context of the nestedconj. For instance, NOT(EXISTS(A, B) & ...) means that if the condition inside the NOT(...) is true for some values for A and B, then the match is discontinued. When that nestedconj is encountered in the match, A and B are assigned the value NIL, and an attempt is made to extend the current assignment to variables including A and B so that the conjunction inside the ()'s is satisfied. The production syntax-checking functions will give a warning if the variables of the EXISTS are used previously outside the nestedconj in the LHS of the same prod. Also, an EXISTS is automatically created (with warning) if a variable inside a nestedconj has not been declared by an EXISTS and if it has not been used outside of the nestedconj context.

rhsconj is used to specify changes to be made to the Working Memory, based on values assigned to variables during the match to the LHS, and using values created by EXISTS, see below. A positive conjunct (without a notchar) is an addition to the Working Memory of a specific instance, while a negative one is a deletion of a specific instance. Note that the Working Memory is fully explicit, containing only positive instances.

EXISTS is used as an rhsconj to specify that new objects are to be created and the objects assigned as values to the variable arguments of the EXISTS. Those values are then used throughout the RHS in building up new instances. A warning is given if an EXISTS variable is used in the LHS of the same prod. Also, an RHS variable not used in the LHS or in an EXISTS is automatically assumed to be an EXISTS variable, and a warning is given. The objects created are based on the names of the variables, for instance, EXISTS(MON, BAN) might create the constants MON-3 and BAN-5. The number used depends on how many previous objects were created using the particular variable. Internally all EXISTS in an RHS are combined into one and put at the beginning of the RHS.

NEGATE is an abbreviation for NOT of those LHS conjuncts referred to by positive numeric arguments, counting from left to right in the LHS. The count also includes entities in the LHS that are not positive conjuncts (a positive conjunct is one that is not preceded by NOT). ALL means that all positive LHS conjuncts are to be negated, whereas ALL followed by negative integers means ALL BUT those conjuncts referred to by the negation of the negative integers. Warnings are given if there is an explicit (using integers) attempt to negate or un-negate something in the LHS that is not positive.

DELAYEXPND is used to cause delayed expansion of a PSMACRO. Ordinarily, PSMACRO's are expanded at SETUP time, thus precluding the dependence of the result on values known only at run time. DELAYEXPND allows run-time expansion to occur, using values assigned to variables during the match or by EXISTS. For instance, suppose STRINGINS is a PSMACRO for the conversion of strings from an external format to a list of conjuncts forming the internal representation. Then STRINGINS('(aa bb cc)) would be a SETUP-time conversion, and the list of conjuncts would be permanently substituted for the occurrence of the STRINGINS expression. On the other hand, if one wanted to insert the internal representation of a string that is computed by the P, one might say, DELAYEXPND(STRINGINS(L)). The macro STRINGINS would be called with L as argument every time the P fired, with possibly a different result each time. DELAYEXPND handles correctly the occurrence of EXISTS in the result of macro calls.

ADDPROD has five arguments: (prodname, prec, comnts, lhslist, rhslist). It is a primitive for adding a P named prodname, with comnt comnts, LHS lhslist, RHS rhslist, and preceding P prec, (if prec is not a P, prodname is taken to be the first P of module prec). ADDPROD causes assertion of (ADDPRODP prodname).

REPPROD(prodname, comnts, lhslist, rhslist, means replace comnt, LHS, and RHS of P prodname as indicated; asserts (REPPRODP prodname).

REPLHS(prodname, lhslist) means replace LHS of prodname as indicated; asserts (REPLHSP prodname).

REPRHS(prodname, rhslist) means replace RHS of prodname as indicated; asserts (REPRHSP prodname).

REPCOMNT(prodname, comnts) means replace comnt string; asserts (REPCOMNTP prodname).

Additional notes:

nestedconj pred's are implicitly locally non-fluent, i.e., Working Memory changes don't result in :SMPX entries for those Ps containing the changed pred's only within the nestedconj context; note that a nestedconj must have more than one element.

Mlisp reference: Mlisp, by D. C. Smith, Stanford AIM-135; copies are available at CMU. Recourse to that should not be necessary for reading programs, or for writing simple ones, especially if study is made of existing examples of PSs.

Prefix operators that are known to Mlisp need not have parentheses around arguments, e.g. NCONS B, CADR X. Binary infix operators may be written between their arguments, e.g. *A CONS B, X @ Y, W NCONC D CONS S (that last becomes (NCONC* W (CONS D S)), i.e., list-building associates right, not left as is customary for arithmetic functions).

Mlisp reserved words: BEGIN NEW SPECIAL END IF THEN ALSO ELSE FOR IN ON TO BY DO COLLECT UNTIL EXPR FEXPR LEXPR MACRO DEFINE LAMBDA OCTAL WHILE STR STRP STRLEN AT CAT SEQ SUBSTR PRINTSTR. Also, the Mlisp translator may balk if standard Lisp functions' names are used in what it sees as illegal contexts.

Mlisp expressions are very similar to Algol, with the feature that functions used are Lisp functions. For instance, + for PLUS, / for QUOTIENT, and so on. Certain characters have special meanings: @ for APPEND, <a, b, c> for (LIST A B C); +, *, /, -, =, ← with standard meanings; logical connectives as mentioned in the grammar above; square brackets are used for list accessing, e.g. a[3] is (CADDR a).

quoted Lisp expressions are Lisp s-expressions preceded by ', e.g. '(EQUAL (PLUS M N) (SUB1 L)); note that Mlisp conventions should be followed when including those in the Mlisp versions of systems, namely, special characters in atoms must be preceded by ? ; internally in Psnlst, the ?'s are dropped or changed to /'s.

Appendix A. Short Summary

A.1. System architecture and production format of Psnlst

A production system (PS) is a set of conditional rules, productions (Ps), that represent changes to a symbolic model of a situation along with conditions under which those changes are to be made. A production system architecture (PSA) provides: a Working Memory (WM), which contains symbol structures representing the dynamic state of the situation being modelled; a Production Memory (PM) which contains the Ps; a particular control mechanism known as the recognize-act cycle, by which Ps are repeatedly executed or fired - a P that is recognized to have its condition satisfied with respect to WM contents is fired by having its actions performed, whereupon the cycle is repeated using the new contents of WM (WM is updated by the actions of the P that is fired); and a set of conventions or ordering principles by which a single rule may be selected from the set of rules that are recognized to be satisfied by the contents of WM during any recognize-act cycle.

The Psnlst (PS analyst) is a PSA, as follows. WM is an unordered set of data items called instances. Each instance is an ordered list of two or more elements, where the first element is a member of a set of constant atoms called predicates, and where succeeding elements are either atoms or list structures - list structures however are opaque, their internal structure not being accessible to the recognition mechanism of the PSA. Instances are considered to be grouped together in the WM according to their predicates. PM is an unordered set of Ps, each consisting of a left-hand-side or LHS (the condition part) and a right-hand-side or RHS (the action part). The form of LHSs and RHSs will be discussed below. The recognize-act cycle consists of a match of the LHS to WM, resulting in bindings for variables contained in elements of the LHS. A firing then uses those bindings to create WM instances according to the elements of the RHS. Two features of the match are unusual. First, all possible matches are found, and a firing occurs immediately for each match. That is, within a single recognize-act cycle, many firings of the same production may occur. Second, a match must include at least one data instance that is new with respect to the P that is matched, where new is defined as having entered WM after the previous firing of the P. The action part of a recognize-act cycle consists of adding or deleting WM instances, and of optionally making changes to PM using ADDPROD and other special operators explained below.

The way Psnlst orders satisfied Ps to select one for firing (this is the fourth PSA component) is by ordering events that occur during the action part of the recognize-act cycle. This is done by using a stack memory that records, for each WM change, the set of Ps that might become satisfied as a result of the change. The stack memory is called :SMPX, stack memory for production examinations. More recent WM changes are stacked on top of older ones, so that Ps satisfied by more recent changes are guaranteed to fire, if satisfied, before Ps using older changes. The order of recency of changes with a P firing are determined by the order of conjuncts within the P's RHS. This ordering principle leaves two selection orders unspecified: if more than one P using the same WM change is satisfied, one is arbitrarily chosen to fire and the other is pushed down in :SMPX by the changes made by the selected P; if a P fires more than once in a recognize-act cycle (more than one match is found for the P), the firings are done in an arbitrary order. With respect to the former arbitrary choice, if one P is to be selected before another one that uses the same WM change, the LHSs of the two Ps must explicitly be mutually exclusive. That is, it is the user's responsibility to distinguish between don't-care and necessarily-ordered situations. Given the :SMPX mechanism for ordering P firings, the recognize-act cycle can be summarized as follows: a change occurs to WM, resulting in :SMPX entries; starting from the top of :SMPX, Ps are matched until a P condition is found to be satisfied; the actions of the satisfied P are executed, resulting in stacking up new entries in :SMPX; and so on.

The following is a Psnlst production that appears in a PS that models a hungry monkey in a room with some bananas, as the monkey recognizes its hunger and tries to reach for the bananas.

```
H1; "HUNGRY" :: HUNGRY(M) & ISMONKEY(M) & ISBANANAS(B) & LOC(B,X,Y,H)
     => GOTO(M,X,Y) & REACHFOR(M,B);
```

The name of the P is H1, its comment is "HUNGRY", and the remainder of the P gives the LHS and the RHS, separated by "=>". The LHS is a conjunction of templates for WM elements; each template is a predicate followed by a list of variables. When a match succeeds, each variable is bound to a specific token from the WM instance corresponding to the template. H1 would match a situation in which the instances (ISMONKEY MNK-1), (HUNGRY MNK-1), (ISBANANAS BAN-1), and (LOC BAN-1 I-1 J-3 K-2) are present, to produce two new instances, (GOTO MNK-1 I-1 J-3) and (REACHFOR MNK-1 BAN-1), assuming, say, that the (HUNGRY MNK-1) instance is a new one. M is bound to MNK-1, B to BAN-1, X to I-1, and so on. MNK-1 is a token for the monkey, BAN-1 for the bananas in the room, I-1 for a spatial location along the X coordinate axis, and so on. The GOTO and REACHFOR instances become instigators of further action, if Ps to model the corresponding real actions exist and if other conditions in the model are appropriate.

A.2. Features of Psnlst programs

The notation for Ps in Psnlst is a subset of the Mlisp language, or rather a special interpretation of Mlisp expressions (see Mlisp, by D. C. Smith, a Stanford AI Lab report, available at CMU). A PS consists of one or more modules, each of which is represented as an Mlisp EXPR consisting of a BEGIN ... END block. Each module consists of optional declarations, followed by a list of labelled Ps. A P is simply a disjunction of an optional comment string and two conjunctions, the first conjunction being the LHS, the second, the RHS. A special function is used to translate these conventions into the format used internally by Psnlst.

The following presents novel syntactic features that are encountered in reading Psnlst programs:

%	- the Mlisp comment character; text between %'s is ignored.
'	- used to quote Lisp S-expressions
"	- string constant delimiter (for instance, Psnlst comments)
;	- a semicolon is used after a P name and to separate Ps
=>	- this symbol separates LHSs of Ps from RHSs

::	- used to separate Psnlst comment string from associated LHS (is DEFINE'd to be OR)
?	- Mlisp character-quote character; must be used for characters that have special Mlisp meanings. For instance, V?-1 is an identifier, not "V minus 1".
&	- AND
<>	- Mlisp syntax for (LIST ...), the Lisp list-building function
@	- Mlisp syntax for Lisp APPEND function, for joining two lists

Summary of notation for Ps:

name ; "comment" :: LHS => RHS ;

The following comments explain other special features of Psnlst programs, but only to the extent necessary for easier reading of the programs. Examples of these features are to be found by the reader in specific PSs.

Macros: certain things that look like predicates are really macros, expanding into a sequence of predicates with arguments; these are usually expanded at load time, by user-defined Lisp programs.

NOT specifies "absence of" when it precedes LHS conjuncts; it denotes deletion when it precedes RHS conjuncts; in LHSs it may also precede a nested conjunction, NOT(...), in which case the conjunction is matched as if it were an LHS, and if it succeeds the LHS match fails; these negated conjunctions may be nested, that is, they may contain nested conjunctions (see also EXISTS, below).

NEGATE is a built-in macro that specifies which of the LHS conjuncts are to be negated in the RHS, by number, or by using ALL; if negative integers follow ALL as an argument, it means "ALL but" the instances specified by the negative integers; for instance, NEGATE(3) would stand for NOT ISBANANAS(B), in the above example.

SATISFIES, SATISFIES2, SATISFIES3 are special predicates for testing values of variables during the match, using Lisp predicates; the numbers 2 and 3 are the number of variable arguments (SATISFIES takes one).

VEQ(x,y) is equivalent to SATISFIES2(x,y,x EQ y), i.e., equality.

VNEQ(x,y) is equivalent to SATISFIES2(x,y,x NEQ y), i.e., inequality.

Conjuncts in RHSs may use arbitrary expressions as arguments, to be EVAL'd as Lisp expressions during the P firing process. (Mlisp includes Algol-like arithmetic expressions.)

NONFLUENT(p) declares p to be a non-fluent, that is, an insertion of an instance of predicate p into the Working Memory does not cause any Ps to be matched for possible firings keyed to that insertion. In other words, no entry is made to :SMPX for that change.

REQUIRE(a,b,c,...) declares that a,b,c,... are required modules of the PS whose main module contains the declaration.

PSMACRO(f1,f2,...) declares files to be read to define user macros.

DCMD(f1,f2,...) declares files to be read as command (CMD) files.

EXISTS in an RHS causes creation of new objects whose names are extensions of the arguments of the EXISTS; those objects are then used in the remainder of the RHS to form instances.

EXISTS in an LHS must be in a nested expression of the form NOT(...); its function then is to locally declare its arguments as variables, causing them to be initialized to NIL for the match that follows, within the (...).

DELAYEXPND(x) where x is some macro call: this specifies that the macro is not to be expanded when the P is inserted, but during the actual firing of the P; this is only used when the predicates of the RHS depend on values not known until run time; it can not appear in LHSs.

ADDPROD(prod,prec,comnt,lhslist,rhslist): primitive for adding a P (named prod) with comment comnt; lhslist and rhslist are lists representing new LHS and RHS; the prec argument is either a P name, indicating that prod is to be placed after it, or is taken to be the name of a new module of which prod is the first P; ADDPROD causes assertion of (ADDPRODP prod).

REPPROD(prod,comnt,lhslist,rhslist): replace comment, LHS, and RHS of prod as indicated; asserts REPPRODP(prod).

REPLHS(prod,lhslist): replace LHS of prod as indicated; asserts REPLHSP(prod).

REPRHS(prod,rhslist): replace RHS; asserts REPRHSP(prod).

REPCOMNT(prod,comnt): replace comment string; asserts REPCOMNTP(prod).

A.3. Features of the trace output

TOP LEVEL ASSERT - the initial starting assertion, typed by user.

! - a P fired

number following ! - the firing was the number'th

P-name followed by '-' then number - the number'th firing of the P

"string" - the comment string associated with the P

USING ... - instances from the Working Memory used in matching the LHS

(xxx . yyy) ... - assignment that was made for the match: xxx was assigned the value yyy, etc.

INSERTING ... - the insertions and deletions made by the RHS

(:SMPX .... number) - a display of :SMPX after firing; number is length of :SMPX; each entry is enclosed in []'s

EXAMINING ... - gives the name of the P and the key insertions causing the examination

/TRY - means that a non-fast-fail examination is being done; fast-fail is a quick check on whether any positive predicate has no instances, before the full-fledged match is tried (formerly /NFF)

WARNING ... - appears when an instance is inserted or deleted but was already present or absent, respectively

*+ - appears for a warning for an instance insertion

*- - appears for a warning for an instance deletion

If the RHS included ADDPROD, REPPROD, REPCOMNT, REPLHS, or REPRHS, a message is printed before the INSERTING line.

PSBREAK comment AT ... - a break in execution; user interactions consist of commands in ()'s; the system responds with output dependent on the command, or with "ok"; (OK) is typed by the user to resume execution.

The above appear on a full :DVERBOS=4 or :TVERBOS = 4 trace; the following are modifications for lesser traces:

the P-firing message is all on one line
most of the EXAMINING message disappears; only the P name remains; if /TRY occurred, only the / appears (in case of verbosity 1, not even P names appear)
most of the WARNING message disappears - only the *'s remain
the USING and INSERTING lines disappear
the messages from ADDPROD et al drop out
break messages, commands, and possibly their outputs disappear

After execution, typically a DUMP occurs (delimited by "DUMP"), followed by the output of PERFEVAL:

Run time for the present RUN invocation
A small table of figures:
EXAM is the number of examinations of Ps
TRY is the number of non-fast-fail (/TRY) examinations
FIRE is the number of P firings
WMACT is database (Working Memory) actions: insertions + deletions
E/F, E/T, T/F give ratios of the first three
the line following the numbers gives an average time figure for each of the relevant numbers in the preceding line (divides total run time by each of the numbers)
Detail on Working Memory changes; "NEW OBJECTS" are those created by EXISTS
Maximum length attained by :SMPX
CORE gives current available LISP core, plus amount used in current run
:ACTS - a list of the major actions in the current core-image
TRACE - a list of Ps that fired, in the order that they fired
FIRED x OUT OF ... - gives number of distinct Ps that fired

Appendix B. System File Pointers

This appendix and the next one are files that are kept on DSK: on the CMU-10A computer under account [C410MR05]. This chapter is not intended as a reference for a Psnlst user - PsnRef.Doc serves that purpose. In many cases the files mentioned in these appendices are not on DSK:, but are kept on backup tapes. The interested reader should request the author by mail that they be made available on DSK:.

Files relevant to Psnlst = Psnlst.Hlp

DSK: = [C410MR05]; unless otherwise marked, files are on dectapes, namely, MR05 MI2, PS2, SV2, and SV4 - file DSK: DTADIR contains directories for those tapes.

Psnlst.Sav	runnable core image for the interpreter (DSK:)
Psnpre.Sav	runnable core image for the pre-processor (DSK:)
Psnlst.Doc	introduction to Psnlst
Psnref.Doc	reference manual for Psnlst
Psnsho.Doc	a short form of Psnlst documentation for non-users
Psdoc.Tdo	*minor additions to system and documentation,* updates to past and present Psnref.Doc
Pstask.Doc	current set of complete or near-complete PSs (DSK:)
PSMisc.Com	several small PSs - see Pstask.Doc
Pstask.Tdo	set of tasks under consideration
Psntst, Psnts2, Psnts3	command files which constitute a test run for debugging Psnlst; - full description of test protocol is Psntst.Alc - outputs from past trials are Psntst.tr?, Psntst.Db?
Psnlst.Alc	some sample allocations of core for P S's - this is now in Psnmis.Com, a combination of files.
documentation	the entire documentation for the system: Psnlst.Doc, Psnref.Doc, Psdoc.Tdo; Lisp 1.6 (doc room), Ilsp manual (see sys: Lisp.Doc); Mlisp manual (doc room, under D. C. Smith); Lisp.Log, Ilsp.Log (minor details on current Lisp); CMU Introductory User's Manual; PDP-10 Monitor Reference manuals

Appendix C. Tasks to Date

Production systems in PSNLST = Pstask.Doc[C410MR05]
UPDATED 22 July 76

General comments:
1. If these are not on dskc or dskb, search file DTADIR, which contains directory listings for all MR05 dectapes.
2. The older ones will not necessarily be up to date with Psnlst.
3. Systems are given in chronological order, most recent first.
4. Naming conventions:
    all files related to a system have names with the same first three letters as the main system name;
    the extension COM stands for a combination file; if the file is large, then it is the main source program plus others; otherwise it contains miscellaneous related files;
    extension ALC gives core allocations for typical runs;
    extension TRS, TRI, TRJ give behavior traces of various forms;
    <three letters>XR.TRS is the usual name of the cross-reference;
    <three letters>C and extensions of that are command (CMD) files;
    extension CTL is a batch control file;
    <three letters or more>M is a PSMacro file;
    DEM stands for demonstration.
5. Psnlst.Hlp contains pointers to Doc's and other system files.

WBLOX + MILIPW - a blocks problem-solving system similar to Winograd's.
KPKEG - King Pawn King EndGame.
GPSR - GPS revisited (Ernst & Newell, 69); GPSTH gives extra Ps for Tower of Hanoi; GPSMC0 & GPSMC1 give two versions for Missionaries & Cannibals; GPSMK is Monkey & Bananas.
Miscellaneous: PSMisc.Com has three files combined on it:
    CRYXYZ is a PS to solve XX + YY = ZYZ (cryptarithmetic);
    Semnet.Mai is some comments on puzzle-solving and semantic nets;
    Resolu.Mai is comments on resolution theorem-proving.
TICTAC - simple TicTacToe, based on Human Problem Solving version.
MONKEY - Monkey and Bananas, written up in Psnlst.Doc.
EPAM - EPAM, adds Ps to represent learned nonsense syllable pairs.
STERNB - simple Sternburg task, variable size, positive response bias.
MILIPS - extension of MILISY, the CMU mini-linguistic system.
STUDNT - Bobrow's STUDENT, for solving high school algebra word problems.
BFGPH - breadth-first graph search (PSMacro file GRAPHM).
PSPCTP - PS for PCTP, a PLANNER program from MIT.
SEGMNT - scan English input and segment according to certain words that are very common, deducing parts of speech (goes with STUDNM).

Chapter VII

Conclusion

Programming with Production Systems

Abstract. This chapter first summarizes the production systems (PSs) implemented for this thesis, reviewing their contribution to knowledge of PSs, their contribution to knowledge of tasks, and the open questions that they raise. Then PSs are evaluated with respect to a number of attributes, among which are practical feasibility, power, overhead, and architectural flexibility. A taxonomy of control is used to highlight the power and overhead aspects. Support is given for the suitability of PSs for understanding systems, by evaluating them with respect to a number of other attributes, with emphasis on modularity and openness. A taxonomy of representation is developed as a means for measuring modularity, supporting the taxonomy of control, and providing openness. At a more abstract level, the methodology of this thesis is examined for its central themes. A sketch of a theory of AI programming is put forward, with preliminary support drawn both from an abstract correspondence to PSs and from the satisfactory concrete realizations of the systems as PSs. PSs presently have some defects, believed to be correctable, and some promising features to be explored, and are at a stage of development where serious applications can be undertaken.

Table of Contents

For Chapter VII

A. Review of the Body of the Thesis

A.1. Review and summary of specific implementations

This subsection reviews each of the PSs implemented, emphasizing the nature of the task performed, the phenomena exhibited by the program, the organization and unusual features of control in the PS, how it contributes to knowledge about PSs, what is gained from comparisons to other implementations of the program, how the PS contributes to task domain knowledge, disturbing and promising features of the PS, and open questions. Motivations and general references for the tasks are given in Chapter I and will not be repeated here. Each chapter starts out with a more detailed description of the task, and includes a fuller set of references, in case the reader needs a brief review of the task beyond the sentence or two given here. Certain topics, such as how control is achieved in the PSs, will be avoided in the following brief summaries, and treated as a whole in Section B. Section A.2 summarizes and explains a number of the superficial attributes of these PSs.

Studnt. This PS is not part of the thesis, and has been presented as a separate, self-contained study of PSs. Some of its major implications have already been discussed in Chapter I. The measures and other discussion of this chapter will be applied to Studnt, however. Studnt takes story problems stated in a restricted natural language and translates them into sets of linear equations, whose solution is the solution of the problems. Studnt solves a diverse collection of 27 problems, applying both general parsing methods to subdivide sentences into algebraic expressions and specific tricks and idioms to allow identifications to be made between variants of semantically equivalent phrases. The primary means of control and organization is a left-to-right scan of a problem statement, applying at each position in the scan all of the applicable idiomatic transformations, dictionary classes, and other operations. Comparison to the original program for this task shows how subroutines and other kinds of control become more data-driven and keyed to the left-to-right scan, in translating from the original language to PSs. Details of the behavior of the PS make some contact with protocols taken of human subjects solving similar problems, and in particular the Studnt PS is readily seen as working in a problem space, a theory of behavior used in other human problem solving studies. The Studnt PS was analyzed in detail to determine its knowledge content and to study how that knowledge becomes encoded as Ps. That analysis showed promising features of the encoding process, and provided motivation for continuing the study of PSs, in particular for the purpose of exploring the encoding of a wider variety of task knowledge.

Epam. The Epam PS is a relatively small program that learns nonsense-syllable associations by incrementally building up a minimal discrimination network. The network, stored internally as a set of Ps that the program can augment, encodes tests that distinguish the various stimulus syllables and emit memory cues. The memory cues are also distinguished by the network to provide links from the stimuli to the externally-given response syllables. The PS's behavior was tested on several ordinary sets of syllable pairs, consisting of three, seven, and nine pairs, and on a set of pairs representing an ordered list, with the response syllable of each pair also serving as a stimulus for another. It exhibited a number of the phenomena that accompany such verbal learning in humans and in previous versions of Epam by other researchers: stimulus generalization, response generalization, and forgetting (retroactive inhibition). (Epam was used to explore using a PS to build a PS, and not necessarily to imitate past work.) The learning in Epam is predefined and rigid, and mistakes occur only in controlled ways.

The PS is organized functionally (not by a structure imposed on P Memory) into an executive to control the input of stimuli and the output of replies in the proper sequence, and to control the other parts of the PS, which test results and engage in corrective modifications and additions to the network Ps. Epam's primary contributions to PSs are its mode of representing the discrimination network and its exploration of operators that a PS architecture should include to allow addition and modification of Ps. Each path of tests from the "top" of the network to a terminal that produces an internal memory cue or a reply is encoded as a P, with the match to the P's LHS corresponding to the traversal of the network. This representation and its manipulation is important because it is a means to storing and effectively using knowledge about specific facts of all kinds, e.g., problem-solving knowledge states, objects that are part of a world model (scenes, faces, etc.), and linguistic knowledge. The particular mode of storage emphasizes its accessibility by recognition of a few distinguishing characteristics in a (possibly partial) description. The results with respect to important PS operators for adding and modifying Ps will be included later (Section B.3). Epam also brought up issues with respect to the architectural alternatives available in PSs for storing information: Working Memory versus P Memory. Within the Epam task, it is quite feasible to store longer-term information, including information on the Ps themselves, as Ps, and restrict the use of Working Memory to shorter-term information - a usage that corresponds generally to current PS models of human information processing. The PS implementation also raises some purely task-specific questions such as whether to include general tests in the network as well as specific ones, whether to make use of possibly-erroneous information from older tests in constructing new ones, and on the format and completeness of internal memory cues. A tradeoff occurs in Epam between being able to examine existing P conditions and storing information about the intent of a P in some other form.

The Epam PS in Psnlst was compared to another version, Waterman's EPAM2, which is coded in a PS architecture that has an ordered P Memory and an ordered Working Memory. EPAM2 has about half the number of Ps that Epam has, and this difference is accounted for in part by the use Waterman makes of the ordering between the Ps his PS adds to represent the network - the ordering allows old Ps simply to be masked out by (placed lower in precedence than) new ones, rather than having to modify old Ps so that they are consonant with new ones. Though a large part of the difference between the two PSs is thus due to using order in adding new Ps, a majority of the difference is due to task-related design features. For instance, EPAM2 uses two distinct networks, one for stimuli and one for internal memory cues, whereas Epam makes Ps in a single net serve both purposes. Because of this, EPAM2 can't learn lists of syllables (as opposed to pairs). Also EPAM2 stores extra information in the network, avoiding tests for compatibility between an incoming stimulus and the stimulus that the network tests were originally built up to discriminate (the two often differ because the net only includes partial tests).

Epam demonstrates the feasibility of a PS that augments itself, but its features raise questions concerning flexibility, generality, and plausibility. It is specialized to the three-letter syllable domain, and this is now seen as a defect with respect to the simplicity of the program itself: a more general program would be simpler, according to preliminary analysis (this conclusion is specific to EPAM, and the "more general" refers only to position dependency of letter tests). It is also specialized with respect to error in the input and to other variations in the task. Such anomalies quickly lead it to construct a network that cannot be properly corrected. The PS is tightly designed, rather than being open readily to modificiation, so that it is hard to envision how the program could be learned in a loose, adaptive way from a more primitive basis. This is not seen as a defect in the PS architecture, but in the present implementation, which is in a sense optimized to work on a particular unvarying task - further research in reformulating it more generally is expected to alleviate this problem. In fact, part of the next PS to be discussed is an object canonization process that does Epam-like things in a somewhat improved fashion.

GPSR. This PS embodies a general problem-solving executive, a number of problem-solving methods, and a variety of other task-independent mechanisms. In combination with a problem specification, also expressed as a PS, it becomes a problem solver with considerable generality and reasonable power over a variety of puzzle-solving tasks. The basis for the problem-solving methods is means-ends analysis, which uses a description of the difference between its current state and the desired state (problem solution) to guide its behavior.

The problems given to GPSR all bring out its basic features: it achieves the combination of a set of general, task-independent methods with very specific problem information. The problems all involve heuristic search in a space where the possibilities are much more numerous than the possibilities actually examined by GPSR before it finds a solution. GPSR solves three problems: Tower of Hanoi, Monkey and Bananas, and Missionaries and Cannibals. The program that GPSR mimics was exercised on eleven tasks, but the three chosen here are representative of most of those eleven and also are varied in difficulty and in the mechanisms used. The Tower of Hanoi task involves moving a stack of disks from one peg to another, with restrictions on how the disks can be arranged on top of each other, using only one intermediate peg for temporary partial stacks. GPSR solves this without a single extra move, a result that derives in part from a fortuitous way of expressing the differences with which the means-ends analysis works. The Monkey and Bananas task involves a monkey trying to get to some bananas, placed outside its immediate reach, by moving a box and climbing onto it. One formulation of the task illustrates some of the chaotic behavior that can result when GPSR doesn't make the right kinds of means-ends connections between differences and the actions taken to remedy them. A more exact formulation (giving GPSR more to work with rather than modifying GPSR itself to act more appropriately) allowed a simpler and more direct solution. Whereas the Tower of Hanoi involves only one kind of problem operator, moving a disk from one peg to another, the Monkey task involves selection from among a set of operators with varying effects. The Missionaries and Cannibals task involves moving three missionaries and three cannibals across a river in a small boat, and is distinguished by having more complex restrictions on how the moving can be done than in the other two tasks. Also the search space and consequently the problem-solving effort are greater. Thus this task was useful for exploring various options in GPSR, for illustrating the weaknesses in some of its methods, and for comparing GPSR to the original non-PS versions of the program. GPSR has a lot of similarity to the original, with identical strengths and weaknesses, but its detailed behavior differs because of different ways of making certain arbitrary choices.

GPSR is functionally decomposable® into a problem-solving executive, various problem-solving methods, and a number of lower-level processes, which are used by specific task operators to perform symbolic operations. Problem states are represented as tree-structured objects. In order to access parts of these objects, advantage is taken of the power of the LHS match, both within the general processes that examine and evaluate differences between objects (using new Ps created as the problem-solving progresses) and within the task-specific Ps (encoding problem information directly in the LHSs of task Ps). PSs are ideal for representing networks of decisions, as occur in the selection of methods, in canonization networks and network schemas, and in making connections between differences and operators and between differences and their estimated difficulties. The match of problem states to each other, to determine the direction to be taken, is encoded as a set of Ps that fire (essentially) asynchronously in a scatter fashion, each pursuing branches of the tree-structured objects and producing differences accordingly. Goal contexts and control contexts within methods are all kept openly in the Working Memory, and no restriction in capability is inherent in this practice (as has been claimed by critics of PSs). PSs also prove effective for a number of processes of selection and generation. Several places in GPSR illustrate how advantage can be gained from the ability to express things either in Working Memory or P Memory. It was found to be quite easy to make alterations and extensions of GPSR in order to test various problem-solving options. An analysis of the knowledge encoded in the executive, combined with some history on how the executive was formed as experience with GPSR was accumulated, shows the ease and directness of augmenting and debugging knowledge.

GPSR raises questions for further research both in the task of constructing general problem solvers and in the use of PSs. It has proved to be a useful and flexible tool for exploring various options within means-ends analysis and related methods, and for trying out variations in the executive. It looks promising for the expansion of the application area of GPS techniques. In particular, the ease in working with GPSR makes possible its use as a general language-system-like basis from which to start, in building specific problem-solving systems: PSs are amenable to task-specific modifications and specializations, and the GPSR concept of an executive distributing problem-solving effort and coordinating communications among a set of loosely-connected methods provides a suitable control organization, and one that is closely linked with the spirit of PSs.

GPSR illustrates a number of difficulties with PSs, but there is also evidence, from representations used, that they can be alleviated within PSs (actual demonstrations must await results of further research). The efficiency of GPSR was barely tolerable, but within an order of magnitude of good performance for any underlying language. This can be traced to two features of GPSR: the number of P firings and the size of Working Memory. The P firings problem could be eased considerably by collapsing adjacent firings, in a number of well-delineated cases, so that the PS would be tuned specifically to problems. This collapsing is possible because the particular adjacent firings are just "interpretive" segments of an operation that could be combined into a single LHS - the general nature of

® The decomposition is determined by the contents of Ps, not by a structural division of P Memory such as subroutining.

GPSR dicates that things be done a step at a time rather than realizing the savings available from assuming a particular object size and form. It is proposed for the future that some mechanism be included to allow this tuning to be done dynamically. The Working Memory size problem is resolvable, according to a post hoc analysis, by making much more use of Ps as a storage medium rather than letting, e.g., goal-specific data just accumulate in Working Memory with the gradual effect of making the match to LHSs of Ps perform more search among irrelevant possibilities. There is also an architectural solution to some of the problems with erasure, namely not allowing Working Memory to expand without limit, but to have, for instance, a fixed size or a fixed element lifetime. Some problems with efficiency and with clumsiness in expressing things in the PS language could be alleviated, it is proposed, by making certain operations, such as erasure and the construction of new Ps, the province of special RHS operators, whose best form can now be deduced from existing PS examples (see Section B.3). Overall, then, GPSR is worthwhile in a number of ways, exercising the control and representational capabilities of PSs, demonstrating problem-solving capabilities, and raising problems that will lead to advances in the design of PS architectures.

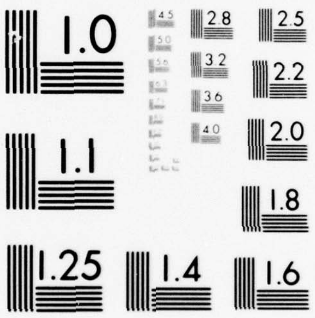
KPKEG. This PS is a limited approach to the domain of chess endgames with two kings and a pawn (king pawn king endgame). Problems in this domain are distinguished among chess problems in lending themselves to solution, as a class, with small amounts of specific chess knowledge and with small amounts of search among possible moves. The objectives of either side are limited: the side with the pawn must try to promote the pawn to a queen and thereby win, and the side with king only must try to block that or achieve a stalemate. KPKEG is an implementation of a strategy hierarchy principle, under which a side first establishes a strategy level and then tries to generate moves that might further that strategy. For the task at hand, there are seven strategy levels, ranging from direct promotion of the pawn or capture of the pawn, through moves to control the pawn's path to its queening square, to moves that try to force the enemy king to back off, and other last resorts. These levels are so arranged that moves generated in accord with a lower-level strategy can never be effective against higher ones, so that there is an immediate limit on the moves that are considered. KPKEG correctly solves three simple positions of the given class, one of which was designed to force the program to go through a moderate amount of search, i.e., one not amenable to immediate solution with a piece of basic knowledge. The problem requiring search (of about 40 nodes) was used to explore several options in using the strategy hierarchy knowledge, including an option of having the program store winning positions and moves in Ps, for future use. The other two tests demonstrate application of various pieces of knowledge not used in the first, each searching less then five nodes before arriving at a solution. These experiments illustrated the ease with which exploration among possible program designs can be carried out with PSs.

KPKEG is organized as groups of Ps representing a strategy executive, strategic evaluators, means to strategies, move generators, and board updaters. The organization among the groups of Ps is roughly hierarchical, but dynamic control is loose, with sequencing of actions based on strategic elements of a position rather than on a strict control regime. The main element of this looseness is an ability for strategies at one depth of a tree to communicate with several levels of strategies above and below it in the tree. This control is achieved through the global Working Memory. Specific chess knowledge is encoded as Ps that contain a minimum of control knowledge, so that its

AD-A037 843 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PRODUCTION SYSTEMS AS A PROGRAMMING LANGUAGE FOR ARTIFICIAL INT--ETC(U)
DEC 76 M D RYCHENER F44620-73-C-0074
UNCLASSIFIED AFOSR-TR-77-0330-VOL-1 NL

2 OF 2
AD
A037843

END
DATE
FILMED
4-77



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

augmentation is simple and direct, while leaving open the possibility of having, instead of a single P, an arbitrary PS program sequence, evoked in the same way as the single Ps presently there. The use of PSs made building the program easy, with much of it built incrementally, filling in missing pieces manually while focusing on other aspects already coded.

The PS can be compared to a Lisp program currently being developed for the same task. Although the Lisp program, CP, is built around a search scheme different from the strategy hierarchy as used in KPKEG, the two programs are still quite similar in approach and in general aspects of program behavior. The biggest differences between the two are in static characteristics of the programs, in run efficiency, and in openness for extension. KPKEG is roughly a third the size of CP, in length of program listing, and about half in terms of a count of primitives (functions versus Ps). On the other hand, CP runs 3-4 times faster, though this is probably due to the fact that the PS is being run interpretively and to a number of other features that could be avoided by re-designing the PS to be especially tuned to the chess task. The PS is much more open to extension and improvement in behavior because the chess knowledge is under less control, and can potentially become driven by new features of the situation as they arise, in a bottom-up fashion. CP uses pattern-like constructions in Lisp to represent chess knowledge, but these are all strictly controlled in a top-down way, and are even evaluated according to their static program order. These features of CP are likely to be detrimental in situations where key aspects of a position arise unexpectedly, during a search whose objectives are vaguely defined or defined in the wrong direction. The PS includes a number of things represented descriptively, in a way that will become more important as the chess task is made more complex, since the program will have to deal with aspects of a situation that are not easily recomputed or recognized otherwise. This capability has not been demonstrated effectively in CP, while in KPKEG, it arises automatically from the architectural design. KPKEG augments its store of knowledge by adding Ps, a facility not in CP at all. This augmentation is promising from the standpoint of automatic generalization and other proposed operations on Ps (to be discussed in Section E.2). Finally, certain features of PSs make KPKEG a good candidate for some proposed mechanisms such as causality analysis, lemmas, and refutation descriptions, which are put forward as essential to efforts to improve chess-playing programs to a Master level.

MiliPS. This PS accepts restricted natural language sentences and either adds to an internal model of a toy blocks scene or answers queries with respect to that scene. It processes sentences without building a conventional parse tree, relying instead on a more direct mapping to an internal representation that is mostly semantic. Inputs containing apparent ambiguities, redundancies, and inconsistencies are correctly interpreted. The referents of complex phrases in the input language are determined by close interaction with the internal scene model. The program operates with a single left-to-right scan across an input, with no backing up in the lexical string to handle anomalies, and with only linear backing up in a semantic representation of the input to resolve inconsistencies. The program processes a test sequence of 25 sentences, demonstrating the mentioned features.

MiliPS is organized around the left-to-right scan of an input. At each point, a number of levels of processing can be done, including lexical, grammatical, semantic, and pragmatic processing. The grammatical checks are minimal: each word must obey simple restrictions on the class of word immediately preceding it in the string. The semantic level of processing divides into three levels: resolving local noun-phrase ambiguities, associating noun phrases together according to relations and predicates (phrases may modify nouns that they are not directly adjacent to), and making use of contents of a sentence according to the main intent of the sentence (according to sentence type). The last two semantic levels deal largely with inconsistencies and redundancies. At each point in the scan of a sentence, as many of these levels is applied as possible, in a bottom-up fashion, so that processing is fairly evenly distributed over the words in the sentence and so that surface structure can be quickly discarded.

The primary contribution of MiliPS to work with PSs is the development of an approach to natural language processing that is direct, simple, and uniform over the syntactic, semantic, and pragmatic aspects of the task - all of these to a degree unmatched by other approaches. The present approach, though restricted to a toy domain, has promise because that toy domain includes, if viewed abstractly, primary elements of most other domains: objects, attributes of objects, and relations between objects. Another indication of the generality of the approach is its position with respect to six kinds of completeness: reference, description of new objects, query logic, manipulation, augmentation, and input-output symmetry. MiliPS is complete to a large degree on the first three kinds of completeness, indicating a basic language adequacy. It is augmented with respect to manipulation by the WBlox PS described below. But it fails on the last two kinds of completeness, indicating that work needs to be done to improve its flexibility in adapting beyond its initial capabilities. The system that MiliPS embodies for working out the interactions between the anomalies of ambiguity, redundancy, and inconsistency seems to be a conceptual advance in natural language processing, exhibiting how the use of PSs leads to organization around fairly natural constructs. It also is stated generally enough to be applicable to wider domains. It is hoped that the MiliPS approach to grammar (simple adjacency checks) will carry over to other domains, but grammar becomes so complex in general that only further research can bear out that hope.

Several features of the system as a PS need further experimentation. The model of the toy blocks scene is presently stored in Working Memory, whereas the general PS approach to storing such longer-term information is to use Ps, e.g., as a discrimination network. MiliPS operates by simply erasing most of its Working Memory between sentences, which is not as general or theoretically clean an approach as, say, having it gradually fade or using deliberate deletion processes. In more general tasks, the decision on what information might be useful across sentence boundaries becomes more complex and is not amenable to such a simplistic solution as is presently used. Finally, MiliPS's run speed is a factor of 5 or so too slow, in comparison to other current systems and in comparison to real elapsed time. This problem will probably be treated by general methods of achieving PS efficiency, with nothing inherent in the task to present special difficulties.

WBlox. This PS combines a toy blocks problem solver with an augmented version of MiliPS to make a problem-solving system with restricted natural language input. It differs from MiliPS in being able to perform manipulations such as putting blocks on other blocks, building stacks, compacting the space occupied by a set of blocks, and finding spaces to put unwanted blocks. The program exhibits satisfactory behavior on a set of 27 sentences, which exercise all of the program's capabilities. The intent of imperative sentences is determined by noting inconsistencies with the blocks scene, according to the system used in MiliPS. Using this scheme, apparent ambiguities in commands are readily resolved. The system establishes goals and subgoals in a hierarchy, to break problems down into manageable components, and it can handle And-Or sequencing of goals. It also implements a backtracking scheme that allows it to search through all possibilities for various actions, if necessary, to find a combination of actions appropriate to the task demands. Much of the need for backtracking, however, is avoided by taking advantage of the selectivity inherent in the LHSs of Ps.

WBlox provides a close comparison to a similar program coded in a Planner-like language. The PS compares favorably in conciseness of program listing with the other version, and is within an order of magnitude of reasonable time efficiency. Effective PS versions of all the features in the original language are easily achieved. The PS must use explicit conventions to achieve the backtracking search, but at the same time, since the mechanisms are explicit rather than built into the language, there is more opportunity to improve performance with task-specific knowledge and to control how the backtracking is coordinated. Although WBlox is restricted to being similar to the original version, several features of PSs are promising for extending the program to more demanding blocks tasks. In particular, still more advantage could be taken of LHS selectivity to avoid unnecessary backtracking, and there are a number of alternative ways within the PS architecture for doing the bookkeeping associated with the backtracking, making the PS flexible for extension. Given PS features, effective implementations are easily conceived for current or proposed problem-solving systems. In going beyond WBlox, PSs are promising with respect to the abandonment of strict backtracking for a more flexible search scheme, allowing more accurate diagnosis of difficulties, more specific corrective actions, and better communication between alternative search paths.

A.2. Statistics on the programs implemented

This subsection gives a wide variety of statistics on the PSs implemented. The presentation is incomplete, in that the numbers are not accompanied by a discussion of their significance. In a sense, this subsection could be considered a footnote or an appendix to the summaries of the PSs. Each description of the various tables includes pointers to where the figures are discussed.

The following table gives some static features of the PSs, in columns in the following order: the number of Ps; the number of predicates; the number of PSMacros and primitive Lisp functions; the space used, statically, in thousands of 36-bit words, divided into free space (ordinary list cells) and full-word space (print names and strings); programming time in weeks, in three fields, the actual programming time, the number of weeks of elapsed time in which an "intensive" effort was spent (intensive = 8 hours or more, an arbitrary boundary), and weeks elapsed in which the effort was weak (non-zero but less than 8 hours); and the hours of programming time per P in the system. This table implies that space use ranges from 100 to 150 words per P. The programming time figures will be used in connection with productivity (Section B.1). Note also that the number of Ps versus the number of predicates is roughly a linear function, but no conclusions will be drawn from this (the taxonomy presented in Section C.2 changes the relationship considerably).

	Ps	Preds.	PSMac. + Fcns.	Space (K) free + full	Prg. t. (wks.) act / int / wk	Hrs./P
Epam	41	37	4 + 17	5.9 + 0.3	2.47 / 5 / 8	2.41
WBlox	130	99	14 + 29	20.8 + 0.9	5.40 / 8 / 13	1.0
KPKEG	141	86	10 + 29	18.8 + 0.8	3.49 / 9 / 11	0.99
MiliPS	193	125	3 + 10	17.9 + 1.1	2.60 / 5 / 11	0.54
GPSR	217	167	11 + 34	26.3 + 1.7	6.54 / 14 / 11	1.06
Studnt	260	251	5 + 25	29.0 + 1.7	6.58 / 16 / 17	1.01
MiliPW	278	184	14 + 14	26.2 + 1.6	see WBlox	
M/WBlox	408	269	14 + 31	45.3 + 2.4	8.0 / 13 / 24	0.78

The figures for WBlox are for the WBlox system alone without the natural language (MiliPW) part, except that the programming time combines the time for augmenting MiliPS with the complete time for WBlox. The "M/WBlox" line gives figures for the combined MiliPW/WBlox system. The number of Ps given is for the main PS plus a typical number of test Ps (around 3), not the full set of test Ps (which set was never loaded all at once). The programming time ratio (hours/P) for GPSR, however, includes coding time for 23 task Ps not included in the 217 total, since coding the task Ps for GPSR turned out to be non-trivial. For the other PSs, the test Ps involved simply posing a task in natural language or whatever was appropriate.

The next table gives some dynamic measures of the running systems. The primary conclusion from this table has to do with the efficiency of PSs, which will be discussed further in Section B.5 and Section E.1.

	Run time (min.)	WM time (msec.)	Fire time (msec.)	T/F ratio
Epam	1.25, .31-3.15	145, 119-202	617, 487-880	1.31, 1.14-1.45
KPKEG	12.6, 8.39-20.8	215, 194-239	935, 840-1060	3.27, 3.12-3.48
MiliPS	.50, .26-1.03	115, 95-162	341, 291-463	1.68, 1.53-1.92
GPSR	32.7, 2.04-66.0	163, 96-234	534, 328-768	1.55, 1.20-1.75
Studnt	5.65, 1.3-20.1	131, 88-212	511, 349-824	2.36, 1.91-2.74
M/WBlox	4.93, 1.42-19.5	261, 184-486	912, 608-1950	1.42, 1.18-1.65

Each column gives first an average figure and then a range of values over the tests run. The columns give, respectively: run times for the collection of tests associated with the PS; times for Working Memory actions, i.e., the total run time divided by the number of actions; times for firing a P, averaged similarly to the second column; and the try/fire ratio, which is the number of match attempts made by the system for each successful P firing, on the average. Epam's times are inflated because the system was run with more trace information than the others, probably by a factor of less than 2. The times are all suspect because there was not a concerted attempt to control the amount of free working space available to the running system. In most cases the fluctuation from varying the amount would be small, but in at least the case of WBlox, it is known that garbage collection consumed up to 50% of the run time, a result that would probably vary considerably with the amount of free space.

The following are some more static values, except for the "Fan i/o" column. These show some characteristics of PSs that will be made use of in discussing features of the language that seem critical to its power (see the end of Section B.2), that support a priori properties of PSs (Section C.1), and that say something about the style of programming.

	N.conj.	Sat.	Fan i/o	LHS uses	LHS length
	avg, max	avg, max		avg, max	avg, max
Epam	.195, 2	.81, 3	2.32	3.32, 14	4.34, 8
WBlox	.654, 6	1.65, 10	NA	4.37, 41	5.21, 17
KPKEG	.496, 7	1.19, 7	2.09	8.5, 77	8.47, 22
MiliPS	.347, 3	.32, 3	1.99	3.94, 37	3.57, 9
GPSR	.641, 5	.81, 17	1.75	3.62, 37	4.47, 34
Studnt	.150, 5	.22, 4	1.80	3.40, 112	4.57, 26
MiliPW	.371, 4	.45, 4	NA	3.91, 46	3.96, 14
M/WBlox	.461, 6	.84, 10	1.96	4.29, 46	4.36, 17

The columns in this table, except "Fan i/o", give an average value and a maximum (minima are all 0 or 1). The values of the columns, respectively, are: the number of nested negated conjunctions, i.e., NOT(. . .), per P; the number of SATISFIES constructs per P; the fan-in and fan-out average value over the Ps, which is a count of the number of Ps that dynamically fire before or after a P; the number of uses in LHSs, for the average predicate; and the lengths of LHSs. The fan-in and fan-out (the average values of which are identical) are computed from typical test runs, or from combinations of several runs. The best such measure would include many firings of each P, but that proved impossible in practice. In particular, the figure for GPSR is based on a relatively small test run, so it is somewhat smaller than the typical value.

A.3. Overview of conclusions

This chapter presents conclusions in several categories. First PSs are viewed narrowly as a programming language. Section B gives a number of features of PSs brought out fairly directly by the implementations of AI programs as described in the preceding chapters. In Section C, PSs are viewed in the more general framework of understanding systems, which raises a number of more general issues. Then even more general considerations are raised, in Section D, as we investigate what the present work says about the nature of doing AI programming. Section E points out the gaps in the evidence about PSs, summarizes a number of promising features that could be explored, and discusses the most serious failings of PSs, the specific features that need to be improved before they can be acceptable for wider use. Section E closes with a discussion of possible applications and misapplications of PSs.

B. Programming Language Issues

This section evaluates PSs as a programming language. Section B.1 gives a number of characteristics that are important to the evaluation, and discusses evidence that PSs are satisfying on those characteristics. Section B.2 presents a taxonomy of the control techniques used over the set of PS programs. Techniques are separated into process evocation and data management aspects. Frequencies of usage of the various techniques give some idea of the power of PSs. Section B.3 summarizes the kinds of improvements that are suggested by programming practices in the completed PSs. Section B.4 discusses the *peculiar forms of architectural flexibility* in PSs, and how it affects programmability. Section B.5 examines the various efficiency factors of PSs that vary over the existing set of PS programs.

B.1. General programming language features

The central questions of the thesis with respect to programming language features of PSs can be categorized as follows:

Practical feasibility: Are PSs feasible in practice, as opposed to formally, for expressing significant AI systems?

Style: Where do PSs fall among the various vague labels that are attached to a language to indicate its style? Most languages seem to be among these possibilities: sequentially imperative, functionally oriented or applicative, and pattern-directed. Some classifiers also distinguish procedural versus declarative, or active versus passive.

Degree of theory-bound-ness: How much do PSs force expression into a coherent view of programming, representation, or approach? This can be taken positively, if the theory is deemed useful, or negatively, if the theory is overly restrictive. A related question is whether PS characteristics are evident at large organizational levels, or whether they are used at a lower level to construct other sorts of organizations and systems.

Power of expression: Which common constructs or imperatives are particularly easy to express?

Overhead of expression: Which common usages are awkwardly expressed, tending to interfere with expression of program content?

Productivity: Are PSs easily coded, read, and augmented?

Efficiency: Do PSs incur an efficiency penalty?

Architectural flexibility: Do PSs offer a variety of ways to express programs, ranging along a number of dimensions such as specialization, generality, conciseness, use of memory structures, and efficiency?

Level: Are PSs a high-level language, with power to express significant computations concisely?

It should be emphasized that considerations here deliberately ignore some factors relevant to human effort in programming, since we are more concerned with using PSs in automatic knowledge encoding systems than with some of the finer points of human programming.

Feasibility has certainly been demonstrated by the six PSs completed. The claims by others (see Chapter I) that PSs are unsuitable for a number of AI domains and capabilities have been, to a large degree, refuted. As we shall see below (Section C) the implementations have been carried out without violating the major properties of PSs from the standpoint of building understanding systems. That is, the programs were constructed without resorting to obscure programming tricks and without building up other control structures orthogonal to the PS architecture. Any objections to the feasibility of using PSs now have to be based on objections and limitations in the set of AI systems implemented. Section E.3 discusses a number of possible further explorations that could answer objections to the set of systems.

Experiments are not required to answer questions of style, at least, not to answer them superficially. PSs are firmly in the class of pattern-directed languages. But they are also sufficiently general to allow expression of programs at the opposite extreme, namely, as sequentially imperative programs. That is, Ps could be arranged to fire in a predetermined sequence, simply by using appropriate data conventions. To verify that the existing PSs are not in fact in that style, we can recall the figures given in Section A.2 for average fan-in and fan-out of P firings. Those figures show that many Ps are followed in execution by a number of other Ps. The average value is around 2, with actual distributions of the numbers of preceding and following Ps ranging up to about 20 for each. (Some of the PSs have over half of their Ps followed only by one P, but many of those same Ps fired only once during the tests on which the data are based.) Although no similar figures are known for other programming languages, it seems clear that a conventional sequential program would not have values much above 1.

On the degree of theory-bound-ness of PSs, Section D is devoted to putting forth a theory of AI programming and to examining how well PSs are suited to the domain as characterized by the theory. That addresses the positive aspects of being theory-laden. To ensure that PSs are not overly restrictive, the negative sense of being theory-bound, it should suffice to point to the wide variety of control and data capabilities that are demonstrated in the PSs. This says little about how the theory affects human usability, since the cost of developing those capabilities is not available. On the related question of whether PS assumptions are evident at larger organizational levels, two kinds of descriptions of the PSs completed give differing answers. First, the PSs are described abstractly as executive + methods + processes + task Ps (GPSR), as a hierarchical set of operators (WBlox), and so on. This is a PS-independent description. Second, the PSs are also described, at a more detailed but still abstract level, using abstract Ps representing varying numbers of actual Ps. The expression of processes in terms of abstract Ps seems to be strong evidence of the permeation of PS concepts to higher organizational levels, while the use of other abstract descriptions is a practice that can be carried out with any underlying architecture, if the description is taken sufficiently abstractly.

To answer questions of power and overhead, Section B.2 develops a taxonomy of control features, and gives some rough measures of PSs relative to it. Evidence developed in the analysis of the knowledge in the Studnt PS will also be used in those measures. An alternative approach would be to apply an abstract model of AI programming, such as the one in Section D, to determine how well the capabilities of PSs are suited to operations put forth as common, by model considerations.

To approach productivity, we can interpret the statistics for programming time given in Section A.2. Programming times range from 2.5 weeks for Epam, at 2.4 hours per P, up 8.0 weeks for the full MiliPS/WBlox system, at an average of 0.8 hours per P (recall that these are actual hours spent, with "elapsed time" respectively of 13 and 37 weeks). Rough estimates place the proportion of designing and coding at between 20% and 30%, with the remainder spent on debugging. The major problem in using these figures for comparison is that such data is not available for other implementation attempts on the same problems. One reason why Epam is high is the difficulty of design by indirection: it is a PS that constructs a PS. Qualitatively, overall coding and debugging times for the PSs seem quite reasonable, and will undoubtedly be improved when there are more efficient PS implementations, since debugging is a major component. Section C.1 discusses the properties of the PSs with respect to augmentation under the topic encodability, which is closely related to productivity.

The efficiency question will be discussed in more detail in Section B.5 and Section E.1. The principal result on efficiency, indicated by the PS implementations, is that less than an order of magnitude improvement in will bring PSs to a reasonable usability. A summary of the ways in which PSs exhibit architectural flexibility is included in Section B.4. These arise in general from tradeoffs between using Working Memory versus P Memory, from degrees of specificity and generality in Ps, and from the varying degrees of use of multiple firings of Ps to perform iterative and other processes. The high level of PSs is supported by their conciseness, which is approached here by measuring the length of program listings, an attribute which affects the manageability of a program while working on it, saying how much of a program can be encompassed visually. From KPKEG, PSs are about three times as concise as Lisp. From GPSR, PSs are estimated to be four to five times as concise as IPL-V. And from WBlox, PSs are roughly the same as Planner in conciseness.

B.2. Control features

A small number of mechanisms of control are used in the set of PSs implemented in this thesis. In reviewing and classifying them here, we wish to get some idea on the overall nature of how PSs achieve control. We also wish to emphasize how few the control mechanisms are. The details of how the techniques are achieved will indicate which features of the particular PS architecture are central. After presenting all of the techniques, frequencies of usage are given, to indicate the power and overhead of PSs, as defined in Section B.1.

Process-evocation aspects of control.

Evocation by a direct signal is by far the most common kind of control used in the PSs. By this, one module, represented by a set of Ps, performs some action and passes control on to another module directly. Control can be passed either by a specific evocation signal or by asserting a result and letting that be picked up by the appropriate successor. Psnlst's conflict resolution process, based on focussing on the most recent Working Memory changes (events), is used to achieve both of these forms. Note that a "direct signal" is not a signal to a particular P, but rather is a goal-like symbol structure, inserted in Working Memory and responded to potentially by a set of targets unknown to the evoker, according to other global conditions. Also, a particular P can respond (be a target) for a number of such signals, as specified by conjuncts in its LHS.

Iteration is somewhat less frequent than evocation, but is still present in a number of places, and in several different forms. The most powerful form of iteration is by repeated firings of a single P. This often performs the function of generating a set of combinatorial possibilities, as is the case in most of the feasible assignment generators in GPSR. It is achieved by having the LHS express a pattern for all of the desired elements to be iterated over (combinations to be generated). When the P is triggered by some signal or new data, it fires once for each such element. A second powerful form of iteration uses a set of Ps but each is still expressed as if only one element were being processed, i.e., with no deliberate looping control. Ps in the set, once started up, fire multiple times, in a scatter fashion, eventually processing fully all the input elements to be iterated over. This is analogous to asynchronous processing in a conventional sense. Since this form of iteration often is used to perform some process on all the elements of a set at the same time, it has been referred to in the body of the thesis as "parallelism". An example of the scatter kind of iteration is the Match-Diff method in GPSR, and an example of the more "parallel" kind of loop is the generation of descriptions of objects in MiliPS. The third form of iteration is a deliberate iteration, using control signals, with explicit testing of completion of the iteration and explicit selection of the element to be used in a single execution of the body of the iteration. An example of this is the stacking of a set of blocks in WBlox. The backtracking mechanism in WBlox is also set up as a deliberate iteration, but one whose execution over the full set of possibilities is rarely carried out.

Processes that are not strictly controlled require coordination (synchronization) mechanisms to recognize their completion and arrange things to continue to further process steps. PSs can do this in two ways. One way is to assert, along with the evocation of an uncontrolled iteration, a second signal that will be lower in priority as an event, and thus whose examination will be postponed until no events in the iteration can be further processed. That is, the method is to make use of PsnIst's :SMPX stacking mechanism for examination of events and their associated Ps. A P that responds to the second signal alone can then assume that the iteration is completed and that the proper continuation can be evoked. This form of coordination is used to make use of the results of the Match-Diff method in GPSR. The second PS coordination technique is to check explicitly for completion each time a result is produced, and if any signals exist that indicate some state of partial completion, re-assert them, and otherwise continue to the next process step. The object-filing process in GPSR uses this kind of coordination to ensure that all objects have been filed when a set of them were input to it. An alternative to these deliberate coordinations that is occasionally used is to let the default processing order take place, using process results whenever they come out, but otherwise just allowing control to fall back to any uncompleted portions when the result-using process can go no further. This is used in the object-description process in MiliPS, and it works because outputs are replies that come at the end of processing of an input, i.e., results that are followed by the awaiting of further user inputs.

Selection is used to perform important functions of narrowing down sets of objects to particular elements, and of deciding how control is to continue. The selection from a set of data items is usually done with a single P, and selection of control is done as a set of Ps. In KPKEG, a single P is used to select the next move to be tried, from among a set of candidates produced by a strategy move generator. That is, the same P always fires, but it produces a selected element according to conditions specified in its LHS. In GPSR, the method selection process consists of a number of Ps, one of which fires to initiate a method appropriate to the goal that is input to the process. The power of PSs in this instance is that one need only specify the cases as separate Ps, with the automatic recognition process performing the selection.

Often a complex process is broken down into a cascade of separate steps. In this way, a decision that could be done as a single P firing from among a large set of Ps is broken down into two much smaller sets. That is, the combinations of conditions are changed from being multiplicative, with each P representing, say, a product of two possibilities along two dimensions, to being additive, with each of two sets of Ps separately making the choices on each of two dimensions. This is done simply by splitting P LHSs into fragments, adding signals to allow the separate steps in the cascade to communicate intermediate results. An example of this is the Try-Old-Goals process in GPSR, which breaks the selection of an old goal to retry into two steps, one narrowing down a set according to a number of criteria, and the other narrowing that result still further to produce a unique selection.

Sequencing of processes involves primarily evoking one process and establishing at the same time a way for things to continue when the results of that process become known. This is done in two ways. The first is to assert, along with a process evocation signal, whatever data is required to combine with the process's results. This assumes further Ps that do the combination and proceed accordingly. The second is to assert a signal that will become active after control falls back from the process, i.e., after no other higher-priority events are in :SMPX. On being recognized, the signal is converted to actions that continue the processing. This second technique guarantees that results are not used prematurely, effectively isolating the process from selections that are to be done on results. Often the signal also effects a renaming of other data that were hidden by a first renaming, to avoid similar unwanted interactions. An example of the first sequencing technique is the sequencing of goals in GPSR, and of the second, the sequencing of steps within the Findspace process in WBlox.

A rough idea of the power and overhead of PSs can be obtained by looking at frequencies of usage of the various features in the PSs. Where counts are given in the following, they are derived from re-examining the PSs, and may not be perfectly accurate, though the general form of the conclusions would not be altered by adding a few missed instances. Counts are based on the static form of the PSs, since we are concerned with programming or encoding properties rather than with dynamic, performance aspects. Evocation by a direct signal is a very heavily-used feature. One measure of its usage is derived from the knowledge analysis that was done on the Studnt PS, where it is evident in more than half of the Ps. Generally in PSs, the distinction between control evocation and other kinds of Working Memory items is difficult to make, since control signals tend to be goal-like rather than goto-like. One criterion for a control signal is that it is used once and deleted, but this isn't always accurate. Among the six PSs, there are 18 cases of evocation by asserting data rather than control signals, which amounts to less than 10% of all control passing. Iteration is used about 60 times over the set of six systems. Of the three forms of iteration, the deliberate form accounts for about half, the single-P form about a sixth, and the multiple-P parallel form about a third. Though the deliberate form is the one that incurs the most overhead and the least power, its overhead aspect is generally not as significant as other forms of overhead, because only a small amount of explicit control is necessary anyway. Also, some iterations are inherently deliberate. Coordination is done 6 times using the "powerful" :SMPX event-order technique, and 9 times using the more cumbersome, deliberate technique. The next subsection will discuss possible remedies. There are 16 instances of the use of selection by a single P. Though this seems a small number, the actual uses made of it exploit its power to a significant extent. The multiple-P, control selection, also a powerful feature, is used in over 20 striking cases and in a larger number of lesser cases, and, along with the *direct* signal evocation, is quite an essential feature of PSs. Breaking down a complex selection or other process into a cascade of steps is used about 3 times. Sequencing using :SMPX occurs about 16 times, while its "check-result" form is used heavily in WBlox goal sequencing and 9 times in the other PSs. Of these two forms, the :SMPX is slightly more of an overhead feature.

To summarize on the power and overhead usages, in the case of iteration, the full power of PSs is not exploited as much as might be desirable, although there are undoubtedly places where deliberate iteration is unavoidable. Coordination presents significant difficulties in terms of overhead, and will be discussed further in the next subsection. The power inherent in selection is well-used in the PSs, and the ability to cascade is not exercised much, but seems potentially useful for more demanding applications. Finally, the frequency of usage of the slightly more cumbersome form of sequencing is significant, though at present is not sufficiently serious to need further attention. Overall, this discussion supports the assertion that PSs are a powerful control structure.

Data management aspects of control.

The above topics have all dealt with a process-evocation aspect of control, but along with evocation, there must be some management of data: operators need to be connected with their operands, and results must be produced and used appropriately. Here, operator is meant in a rather abstract sense, as something (a process, module, set of Ps) that takes some input data (operands) and produces some action or result. In general, the data management is performed simultaneously with the process evocation, with LHSs performing some data connection operations, and with RHSs often combining both process and data actions. The following will discuss several such topics in turn.

<u>Connection</u> is made between operators and operands within the PS match. That is, the match takes an evocation signal and uses data arguments of Working Memory instances to form coherent patterns, which then constitute sufficient context for an operator to be applied. Often, the necessary links are formed between instances by using specific tokens, e.g., unique goal names. Matches can involve following chains of such associations to bring in all of the required items.

<u>Arranging results</u> and result-usage signals takes place within single RHSs. An exception is the operator-application preparation that is done in GPSR, but that is at a much higher level than we wish to examine here. That is, that kind of arrangement is on a different scale. Within single RHSs, all arrangement takes place for communication between operators composed of sets of Ps. The alternative would be the evocation of preliminary setup operations, expressed as separate Ps, but this does not occur.

Renaming of data is used for two purposes: protecting data so that it will not be used in the wrong way, and the converse operation of releasing data for use that had been protected. In the PSs at hand, renaming is done along with other operations within a single P, appearing as a side effect of some other process. It is also done by an explicit evocation signal, to be processed by a single-P iteration (more complex iterations might be used, but no such occur in the PSs here). Often that evocation signal is placed after some other signals, so that it is examined and used after some process, according to the event-order conflict resolution mechanism. Epam uses renaming (of the side-effect type) to save the results of one net-P firing cycle while a second cycle occurs, so that the former results can be properly distinguished from newer similar data.

Cleanup is the operation of deleting or otherwise disposing of old Working Memory instances so that they don't interfere with further processing. It is generally carried out in the same ways as renaming: within single Ps, as a side-effect, and in a more explicitly iterative way. GPSR contains a number of examples of erasure Ps, which fire in single-P multiple-fire loops, for instance, erasing unneeded Match-Diff intermediate data.

A mechanism for handling data that is only slightly used in PSs to date is having information stored as Ps for use at some later time. This is done entirely by deliberate processing, generally including iterations that gather the various components of the Ps to be built. GPSR builds recognition networks of Ps that are then used in recognizing the occurrence of previously-seen problem-solving situations, thus performing indirectly the important control function of preventing repetitions.

To conclude our discussion of the data aspects, we touch on the topics of frequency of usage, power, and overhead. The connection of operands with operators and the arrangement for communication of results are very common operations, and their simplicity of implementation indicates their relatively high power. In this case, the power derives from the basic PS rule form. Renaming is rare, but is also easily achieved and powerful. Cleanup is sufficiently frequent in its less powerful, iterative form to be a significant problem for some of the PSs (GPSR in particular). The need to go through deliberate iterations to store information as Ps is also of an overhead nature, although the use of the stored information later is a powerful mechanism, since it avoids other sorts of deliberate processing. Section B.3 will discuss possible improvements in these last two weaker features of PSs.

Essential features of Psnlst.

Four features of the Psnlst PS architecture are essential for these control techniques. The main one is the use of event order (:SMPX). This has allowed the PSs here to overcome many of the PS control problems that have occurred or have been predicted with respect to other PS architectures or PSs in general. The testing within LHSs of Lisp predicates (expressed as SATISFIES in the language) is used heavily, with about three fourths as many occurrences as the total number of Ps (which is not to say that three-fourths of the Ps actually contain an occurrence). Of the uses of SATISFIES, only a few are used to test equality to a constant, so that most are used for more significant purposes such as testing numerical relations. The allowing of multiple firings on the same recognize-act cycle plays an important part in the power considerations above, being used in a number of ways to implement the more powerful PS control facilities.

Negated conjunctions of condition elements are used overall about .414 times per P on the average, a higher proportion than negations of single elements, for which the figure is .255. Their overall importance is indicated by their frequency of usage and by their use in implementing the control facilities above.

B.3. Suggested improvements in basic operators

A number of improvements in the basic PS operators are indicated by undesirable overhead properties. These were all initially suggested in the various chapters where they first became evident, and some have been reviewed briefly in Section B.2. They are not expected to change the area of feasibility of use of PSs as much as to make them more powerful in locations where some awkwardness has been noticed. In some cases, the suggestions are superficial language changes that would have no effect on present program structures, usually shortening or modifying components of individual Ps, while in others, the changes would be more far-reaching, and are thus perhaps more controversial with respect to keeping PSs simple. The basic aim, however, is to be very conservative and not to go beyond modifications that are suggested directly by the evidence of existing PSs.

The experience with adding Ps in Epam, GPSR, and KPKEG indicates a set of operations for doing so that are less general and more direct with respect to the commonly-used functions within the present P-adding operators. That is, the present general operators - adding an entire P, replacing a P's LHS, replacing a P's RHS, and replacing an entire P - are actually used only to perform a simpler set of functions, with the full generality of those operators only interfering with the functions. Some candidates for additional operators, to be verified with further experiments:

Extend an LHS by adding conjuncts at the right end (not left); in Epam, for instance, this would be used to add tests for more letters, in order to make finer discriminations.

Extend an RHS by adding conjuncts, either at the right end or in a position relative to a conjunct satisfying some pattern; e.g., one might want to add an action before some known action.

Split an LHS into two parts, extending the LHS in two different ways by adding two lists of conjuncts.

Add conjuncts to a nested negated conjunct in an LHS; this was used in GPSR to extend a negative test on an object with further tests, to go along with adding similar tests to the LHS of another P.

Update an RHS conjunct by replacing one constant with another; this is used in Epam, for example, to change the reply image in a P.

Make the contents of the LHS or RHS of a P available for inspection in Working Memory.

Make the variables in a newly-added portion of an LHS be unique relative to existing variables there, to ensure that the new portion doesn't interfere in matching.

These operators and options would have the effect of making the parts of PSs that deal with adding new Ps and refining old ones more concise, simpler in terms of PS control, and simpler with respect to the amount of basic list-processing necessary to form the new structures.

An operation in LHSs that occurs in most of the PSs is essential to selections, to narrowing down the set of possible matches to one with particular properties. Consider as an example trying to select the element from Working Memory with the highest numerical value for some predicate. At present, this would be expressed roughly corresponding to "an instance of the predicate such that there does not exist another instance with a higher value." The effect of this is for the matcher to successively test each instance of the predicate until all the ones with the maximal value are found, with match failure occurring for the others, after an iteration through part of the instances for each. What is really needed here is to be able to apply the "highest value" predicate to the set of possible values for the other predicate, and to *immediately select those with* maximal value. That is, one writes the P knowing that at a certain point there are going to be a number of possibilities, and one wants to express an intention with respect to that set of possibilities rather than to use the indirect present method, which says essentially "and there doesn't exist a match that does any better." In many cases at present, it is necessary to apply several of these tests to narrow down the possibilities, and for each additional one there must be a laborious recapitulation, within the "not exists" nested conjunction, of the restrictions that have been applied so as to ensure that each "not exists" is working with the proper set of possibilities. What is proposed is that instead, when a series of restrictions are to be applied, each applies to the set of possibilities remaining at that point.

Some more explicit examples will clarify these distinctions. Suppose the new primitive is named MAXIMAL, that the evaluation predicates are HIGHER1 and HIGHER2, and that the Working Memory *predicate whose instances are being selected from is PRED1.* Then the expression,

```
PRED1(X) & NOT( EXISTS(Y) & PRED1(Y) & SATISFIES2(X,Y,Y HIGHER1 X) )
```

would be expressed,

```
PRED1(X) & MAXIMAL(X,'HIGHER1).
```

The expression,

```
PRED1(X) & NOT( EXISTS(Y) & PRED1(Y) & SATISFIES2(X,Y,Y HIGHER1 X) )
   & NOT( EXISTS(Y) & PRED1(Y)
      & NOT( EXISTS(Z) & PRED1(Z) & SATISFIES2(Y,Z,Z HIGHER1 Y) )
      & SATISFIES2(X,Y,Y HIGHER2 X) )
```

would become,

```
PRED1(X) & MAXIMAL(X,'HIGHER1) & MAXIMAL(X,'HIGHER2).
```

Examples of tests even more deeply nested than the second one can be found in GPSR, KPKEG, and WBLOX. In GPSR, in the Try-Old-Goals process, a test involving a number of such restrictions is split into two Ps, the second applying to the result of the first, in order to make it possible to untangle all the conjoined tests. The use of MAXIMAL improves readability, is much closer to the intended concept, and should have some advantages for PS efficiency.

· Several changes to the way matches are expressed and to the way actions are specified are suggested by the PSs. Currently, LHS elements are a predicate constant followed by a list of variable arguments, and if an argument is to be matched to a constant, it must be done indirectly through the SATISFIES mechanism, which can in fact test much more general attributes of a value corresponding to a variable. This form, assuming a constant head and variable tail, is in practice almost always the right one, with the average use of equality to a constant being made less than once per P in all the PSs (that is once out of a large number of match variables). Nevertheless, cases have occurred in the PSs where the indirectness of testing equality to a constant seems to incur unnecessary combinatorial matching cost, so it is recommended that quoted constants in condition elements be allowed. This would bring the pattern-matching more into line with all of the other pattern languages in use. The restricted form was tried partially as an experiment, to see just what the consequences would be, and partially because the implementation was simpler. Another restriction that can probably be abandoned is the failure of the match to descend into structured lists. That is, all matching is done only at the top-level argument list. Structure in matching is useful at least in the case where there are a large number of arguments, some of which can be more easily grasped if bound together into distinct sub-lists. This would be useful, for instance, in WBlox, for representing the numerous occurrences of spatial coordinate triples. A third restriction in Psnlst is that predicates must always be constant, where conventional AI usage would dictate a capability for the predicate to be variable, like anything else. In the original implementation of Psnlst, for this feature, efficiency was a consideration, in that having predicates be constants allows the construction of simple indexes of possible uses of new Working Memory elements. But experience with the PSs indicates that there are important gains in flexibility, and perhaps also in efficiency, by allowing at least a restricted form of variable in the predicate position, both in conditions and actions. At present, the most reasonable compromise with the original version is to allow variables in the predicate position that are restricted to being bound to one of an explicit set of elements.

Several other improvements have been suggested by the various PSs. One is a more powerful erasure operator, for instance allowing the deletion of all the instances of some predicate, or of all instances satisfying some pattern. Such an operator would be useful in reducing the number of Ps that currently perform only erasure operations, and is a matter of convenience as much as it is an improvement in efficiency. A second facility suggested by some of the PSs is a general set of functions for taking external forms and converting them into usable Working Memory items, and for doing the converse operation to produce externally readable structures from Working Memory elements. At least two kinds of representations would be useful: one for strings of atoms, as are used in Studnt and MiliPS, and one for structured objects as in GPSR. The conversion operators could be very close to the ones developed ad hoc in the mentioned systems, but built into the system and generalized somewhat to be more widely usable. A third new operator would be useful for providing a more powerful means of coordination than the deliberate condition-testing used in the PSs in a number of cases (discussed somewhat in Section B.2). A specific operation suggested is some kind of an explicit delay of examining some new Working Memory elements until other consequences that have occurred since some specified change have been completely finished. This would allow older consequences to take effect before initiating some new process that assumes that all those results are available for examination. It amounts in effect to allowing a P to explicitly place its actions somewhere below the top of the stack of events (:SMPX).

B.4. Additional programmability topics

There are several ways that PSs exhibit architectural flexibility. One is in the variety of implementations of the main control techniques, as discussed in Section B.2. A second group of features revolves around tradeoffs between the use of Working Memory and P Memory. The three different ways to build Epam, as discussed in Chapter III, are based on differing uses of Ps and Working Memory. In connection with GPSR, there is the potential use of Ps to store goal and object information. Ps are used in GPSR to store information about loc-progs, and could be used in a similar way to store information about move operators, with added benefit of providing a better mechanism for noting when proper use is made of that information. In GPSR and KPKEG, a variety of ways of controlling the status of processes to generate sets of possibilities are illustrated, and the discussions of the systems include additional ways. In GPSR, there is flexibility in specifying external tasks to do, in that task specifications can use more information expressed as Ps and less as Working Memory items. The discussion of MiliPS raises the possibility of storing the scene representation as Ps instead of as Working Memory elements. Maintaining trees of alternative data contexts while searching is discussed in connection with WBlox, and a number of alternatives, varying in their Working Memory and P Memory usage, are suggested. Generally, the existing PSs do not exploit the use of P Memory as much as they could, but it is expected that as more general and powerful systems are built, making greater demands, P Memory will be more commonly used.

A few other forms of architectural flexibility are illustrated by the PSs. The discussion of GPSR points out the range of expression used in various networks: some of the networks use very specific constants, others use a few very general Ps, with mostly variables in conditions, and others are expressed as objects, with the tests carried out interpretively. GPSR also has promise with respect to becoming specialized to particular tasks by building additional specific Ps as tasks are manipulated. More generally, the PSs include different degrees of specialization, with the very general GPSR, and the more specialized WBlox and KPKEG. WBlox in particular is specialized to a specific goal-subgoal structure. Finally, the discussion of the augmentation of MiliPS for WBlox points out how the architectural flexibility can be used in dealing with the problem of distinguishing between temporary relations, which are computed as needed and then discarded, and other more permanent data, from which, for instance, the temporary data is derived.

Thus there is a wide range of variability in storing information and in conventions for use of data for communication. Information can be stored in RHSs, available with the appropriately-keyed demand; it can be actively stored as a network, accompanied by conditions in which the information is judged automatically to be useful; and it can be stored as structure that is interpretable when its content is desired. Changes to structures can be stored incrementally so that they can be revoked in the same fashion. Actions can be packaged for unconditional execution in one sequence or can be broken down into conditional parts. A set or its complement can be kept in Working Memory, and similarly, Ps can be set up to record what's left to do (the set proper), or they can be set up to remove possibilities already considered (the active form of the complement of the set). Data can be kept in Working Memory as a means of making it temporary in nature. The deletion of data can signal the proper use of data or the completion of some process. All of these examples illustrate the particular kind of flexibilty that PSs have. Rather than thinking of programming with a completely arbitrary means of representing and processing, there is a predisposition to a small set of powerful mechanisms. The mechanisms derive from the split in the architecture between a relatively small, passive Working Memory, and an active P Memory. There is a range of specificity/generality from concrete Working Memory elements, through slightly abstract P conditions and actions, to more distributed representations, with information encoded, in whatever way is appropriate, as a mixture of evokable data, recognition processes, and general elaborations on those.

B.5. Variations in efficiency over the systems

There are a number of interesting variations in the figures of Section A.2 that bear on efficiency. The figures in question are those for average cycle time (P firing), average Working Memory action, and for match effort as reflected in the try/fire ratio. There is little apparent relation between changes in those values and the number of Ps in the system. The highest values (indicating least efficiency) are attained by KPKEG, one of the smaller systems, and WBlox, the largest. But these systems seem not to be systematically high on other attributes. The apparent lack of trends on a number of dimensions leads to the hypothesis that there are more complex factors that determine the efficiency of a PS, and that these factors combine in some way to make a PS efficient or not. Such interactions will thwart attempts to build simple models of efficiency. Perhaps what these factors are will emerge as more PSs are constructed, providing additional data points.

A few isolated characteristics may have something to do with the variations. KPKEG is high on length of LHS and on the number of SATISFIES tests, so these may combine to make matching more expensive. An inherent property of KPKEG is close similarity between Ps, so that in order to find a match from among a set of Ps, most of which are plausible by superficial criteria, much effort must be expended. WBlox suffers from quite a different problem, apparently. Its size causes it to put unexpected strain on certain implementation deficiencies, particularly those having to do with needless use of temporary list cells. This usage forces an unwarranted number of garbage collections, and the sheer size of P Memory, all of which must be examined during that process, adds significantly to system overhead. As was discussed in Chapter IV, GPSR incurs some inefficiencies by overloading some of the Working Memory predicates with too many instances (which can be alleviated by using P Memory), and also spends a significant portion of time doing cleanup by firing erasure Ps.

C. A Basis for Understanding Systems

This section first assesses the position of PSs with respect to most of the understanding system characteristics laid out in Chapter I. Two of those emerge as a focus for the discussion in the rest of the section: modularity and openness. Section C.2 presents a taxonomy of representation intended to meet possible difficulties of PSs with respect to that focus. Section C.3 discusses the results of applying that taxonomy.

C.1. General features

A number of observable properties of the implemented PSs can give support to the suitability of PSs for building understanding systems. Because no understanding systems of the sort aimed at were built as part of the thesis, the evidence summarized here can only provide evidence for plausibility, not a full demonstration. We will address two sets of related properties, which were introduced in Chapter I. First there are the a priori properties: properties of conditions and actions, properties of how Ps interact, and properties of the overall architecture. Only a couple of these properties need empirical support, the size of conditions and actions, and the degree of interaction between Ps. Over the entire set of PSs, the average of the averages of LHS and RHS sizes (given in Section A.2) are respectively 4.87 and 4.34. These numbers support the small unit size of Ps, and also support the assertion that PSs have a high degree of selection (LHS) for each action (RHS), as compared to other typical AI programming systems. How Ps interact is much more difficult to measure, since it depends on static, organizational, and dynamic properties. Most of this section will be devoted to this topic, but first we summarize some other aspects of PSs.

The second set of properties is the list of properties of systems that are critical to their use as understanding systems. Recall that Chapter I introduced a set of primary properties: encodability, organizability, inspectability, accessibility, flexibility, and operability. These primaries have a number of secondary aspects, many of which are shared among several primaries: modularity, conciseness, uniformity, transparency, provability, explicitness, and openness. There are a few specific points to be made here in support of some of these properties for PSs (and some of them are also supported by a priori properties of PSs, as discussed in Chapter I).

The encodability of knowledge in PSs is supported primarily by the feasibility of the six PS implementations. Some further support is given by observed properties of building and extending those PSs. In GPSR, a number of executive options were added on after the program was working properly. Those options were quite easy to implement, with no unexpected, indirect interactions. Similarly, in KPKEG, there were no complications in trying various options in the search executive. In augmenting MiliPS to work with WBlox, a large number of changes were made, but only a very small number involved changes to existing Ps (adding a condition element or modifying one or two RHS elements) while the vast majority involved simply adding Ps. PSs are different from other languages in having more independent units of augmentation (Ps). That is, usually Ps are just added, with no decision necessary as to where a P goes (since the static order of the PS is irrelevant to processing) and with little attention necessary on how it is to be used, since its condition is explicit. The related property of inspectability of knowledge in PSs is not a problem in practice, although programming by a human and the construction of a process to inspect Ps automatically (as an adjunct to encoding further knowledge) involve different issues. The central problem in automating, with regard to inspectability, will be the lack of openness of the representation, a problem that will be attacked at length below.

The conciseness of PSs is supported by the relatively small number of Ps required for the various programs. For instance, KPKEG is 2-3 times smaller in size of program listing and in number of functional units than a comparable Lisp program. PSs are a particular way of managing data and procedures that suffices where a number of ad hoc mechanisms have to be included in Lisp programs at the expense of conciseness. Transparency of PSs is supported by the Studnt knowledge analysis. That is, the knowledge analysis gave evidence that the Ps are a level of expression minimally interfering with the primary encoding task and also a level of expression close to natural language statements, which in turn are derived from an abstract process model. In the case of Studnt, almost three fourths of the knowledge is about the task and about solving task problems, with most of the remainder devoted to programming techniques and a small fraction devoted specifically to PS control. The flexibility of PSs is evident in the variety of architectural alternatives, discussed in Section B.4. The organizability of PSs is supported by the variety of organizations used in the six programs. GPSR implements a general method coordination and method evaluation problem-solving structure. KPKEG implements a straightforward heuristic search approach. WBlox is organized as a specialized goal hierarchy, with the addition of backtracking as its search method. The MiliPS bottom-up recognition hierarchy is yet another organization. None of these organizational structures was at all troublesome to put together, indicating that PSs are well-suited to a variety of approaches. Some of the organizations are modified slightly and made more powerful by taking advantage of the power of selection of PSs.

Two major areas have been left unsettled by the above discussion: the area encompassing the questions of modularity, provability, and interaction; and the area of openness. Having Ps interact too much with too many other Ps is undesirable in a number of ways. With large PSs, and we fully expect understanding systems with many thousands of Ps, then interactions could get too diverse to be taken into consideration effectively, so that some kind of structural subroutining would have to be imposed. This would subvert many PS properties that depend on having the system be a uniform, single-level structure. Too much interaction implies also that there are assumptions made in writing each P. That is, it implies that there is global context of some sort stated explicitly in Ps. This makes Ps much harder to modify while maintaining global correctness. It gives PSs the overall appearance of being intricately-assembled pieces that are somehow coordinated to drive a sequence of global actions. Also, the provability of correctness of knowledge in a system is much more difficult when there are such far-reaching implications and interactions. The remedy for such objections to PSs is to show their modularity. The representation taxonomy of Section C.2 will provide a means to that proof.

The other major understanding system trait that is noticeably lacking in the PSs implemented so far is openness. These PSs are closed in several senses: they work only on the narrow task domains for which they were designed, with their knowledge encoded in such a way that it would not be applicable unless a number of Working Memory conditions (all associated with the particular task implementation) held; they use a number of internal names that are effectively inscrutable to other processes, so that general processes of error diagnosis and progress evaluation cannot be applied when problems are encountered - if such problems can be recognized at all; and the naming conventions prevent new external knowledge and newly-developed internal knowledge from making appropriate contact and interacting with existing knowledge (assimilation), and thereby also prevent such knowledge from being incorporated on a longer-term basis (accommodation). Openness is the key to allowing a program to respond flexibly to variants of a task, by improving its assimilation capabilities, and thereby is the key to unexpected generality and power. Further, open programs can readily be combined into larger units, with opportunity for sharing processes and capabilities, and for application of methods from one to new problems in others. The representation taxonomy below approaches openness by providing a rational basis for naming. There are processing requirements for openness that go along with such naming, which cannot be specified at present, being more properly the subject of further explorations. It seems at present that representational barriers to openness are much more serious than are any lacks on the processing side.

C.2. Representation taxonomy

The taxonomy of predicates' meanings in this subsection supports three aspects of PSs: their modularity, their openness, and the simplicity of control constructs required - a verification of the control taxonomy of Section B.2. By breaking down predicates used in the six PSs into more primitive meaning elements, and by replacing ad hoc abbreviations and conventions with a more rational scheme, the number of meaning elements is drastically reduced and the interactions of various predicates becomes more transparent. Recall that predicates are the constants that occupy the head position of each Working Memory element and each LHS and RHS conjunct, thus constituting the essential meaning of both dynamic and longer-term structures. Renaming the predicates as proposed here does nothing to computational properties of the present PSs, but provides a potential openness for interaction which future applications will exploit. The following scheme should be considered a first approximation, sufficient for the purposes of this chapter.

The taxonomy divides predicates into two major types: those that refer to processes, and those that refer to data structures. Each predicate is broken down into three components: a primary name, an optional secondary name, and a set of modifiers. The primary name is the main process name or the name of the global data structure being referred to. Primary names are rather general concepts (evaluate, apply, goal, object, network), and although they originate with specific tasks, it is hoped that as a system grows and expands its task domain, there will grow up, around a primary, a useful set of associated knowledge (expressed as Ps). The secondary name is a qualifier to the primary name, in case it has attributes, entry points, subprocesses, case frames, manner qualities, and other such subconcepts. In some cases primary and secondary are verb and direct object. The modifiers are a set of tags that apply to show further subaspects such as truth value and degrees of imperativeness (in a vague, non-technical sense). A global data object such as a goal with a variety of attributes has primary GOAL, with secondaries like actual object OBJECT, difficulty DIFFIC, supergoal SUPER, etc. These would be written as GOAL *OBJECT, GOAL *DIFFIC, and GOAL *SUPER, according to the proposed notation, which places "*" before secondaries. With "/" preceding modifiers as in the proposed notation, and given that "T" is the modifier for "true" truth value, the GPSR predicate HASSUPERGOAL becomes GOAL *SUPER /T. The primary SELECT might have secondaries like OBJECT, GOAL, and METHOD. A concrete object is subdivided according to its attributes. Thus an OBJECT might have secondaries TYPE, SUBOBJECT, LOCATION, and SHAPE.

The main content of the taxonomy, at present, is in the modifiers. (The definition of secondary vs. primary is also content, but is left vague.) Modifiers are in five classes: goal values, truth values, process types, data types, and degrees (see Figure C.1). A modifier has three positions in general, i.e., is composed of at most three things: a goal value, one of {truth value, process type, data type} (a mutually exclusive set of classes of values), and a degree (with possible subdegrees tacked on).

The expository notation adopted here suffices for the purposes of this chapter, but other issues should certainly be considered if a notation is to be used effectively by operational systems for self-examination. That is, what is readable for a human may not be suitable for a PS to use, in both pattern-matching capability and openness. Three dimensions of variation of notation can be distinguished: nested, open structure versus tight encodings as strings; internal versus external modality; and implicit or explicit argument typing.

For human readability, the *first dimension* includes a tight encoding with distinctive characters to segment a string, e.g., EVAL*GOAL/WA.2. For a list-processing-based PS, though, with strings taken as units (atoms), structure must be indicated differently, as in (EVAL GOAL (W A 2)) or ((EVAL GOAL) (W A 2)).

The second dimension deals with the location of the modifiers, with external modality common in some published PSs, e.g., (OLD (RESULT (EVAL GOAL G-3 OK))). In this, arguments have been added to the primary and secondary in order to illustrate a complete Working Memory element. The corresponding internal modality (adopting abbreviations) would represent it as (EVAL GOAL (O R) G-3 OK). Internal modality gives more prominence to the primary and secondary, and makes it easier, in conventional pattern-matching schemes, to have an optional degree position - the absence of a tail of a list as opposed to the absence of a level of nesting.

The third notation dimension deals with whether to have explicit type tags for arguments or to let types of arguments be implicit in the position within the list of arguments. Typed arguments are common in semantic network representations. Moore and Newell's Merlin (1973) uses explicit typing to allow a general interpreter to make mappings between structures, some of whose components are optional or incomplete. Similar advantages are claimed by Hayes-Roth (1974). The element (EVAL GOAL G-3 OK), which uses implicit ordering to type its arguments and to distinguish primary and secondary, might be rewritten (prim:EVAL sec:GOAL goal:G-1 value:OK). One can envision mixed strategies for typing, but wherever implicit typing is used, auxiliary information is necessary for complete openness.

To conclude this brief discussion of notation, the best approach for future work would be to use a representation with nested list structures and with internal modality.

<u>Goal values</u> (modalities):

W	Want, want to achieve, want to activate.
D	Don't want, want to deter or delete or disable.
O	Old, no longer current.
B	Been achieved, "be", a neutral goal status.

<u>Truth values</u>:

T	True or succeed.
F	False or fail.
M	Maybe, in progress, partial.
U	Unknown, but attempt has been made.

<u>Process types</u> (types of imperatives):

A	Activate.
C	Check, combine (as in combining present data with the result of some subprocess), coordinate (as in coordinating the results of several processes or lines of "parallel" execution), continue (after solving a subgoal).
H	Hold (as in holding a signal until some other event has had its chance to go through, whereupon a P converts it to active status).
G	Generate, gather, or more generally iterate.
S	Select (as from a set of similar items).

<u>Data types</u>:

R	Result;
E	Effect, side-effect, error condition or indication, extra information (addition to main result).
X	Conte<u>x</u>t (as for a process).
I	Input (as to a process, in addition to predicate arguments).
K	Knowledge about, information about (knowing about a process is distinct from activating it, for instance).

<u>Degrees</u>:

1, 2, 3, . . . Steps in a process, degrees of completion, degrees of certainty; substeps and subsubsteps could be indicated by stringing together a number of degrees; when strung together, "." is used as a separator, e.g. 2.17.4.

Figure C.1 Values for modifier components

These preferences are based on present pattern-matching capabilities. There is the possibility that, since a lot of list structure is imposed by these preferences, the assumptions should be built into the pattern-matching algorithms to avoid unnecessary condition-testing. On the third dimension, implicit typing seems to involve less symbol-processing, and is thus preferred at the moment, but may become unworkable later because of difficulty in determining the implicit information.

We now use the following P, from GPSR, to illustrate this renaming process.

```
M38; "GEN DES ASG*" :: CHECK:NUMV(DA) & GENDES:ASG2(G,OP,DA,C,L,D)
     & HASVAR(C,VAR) & HASVAR:LINK(VAR,P) & HASLP:COMPON(L,P) & VAR:DOMAIN(VAR,N)
     & SATISFIES2(N,D,NUMBERP N & N GREATERP 0 & NOT(N GREATERP D))
   => ERASE:LPC('T) & FILE:DES:ASG(DA,OP) & FEASASG(OP,DA,G) & ASSIGNS:N(DA,VAR,N)
     & NEGATE(1,2);
```

This P is from the process that generates desirable assignments for move operators, a part of the Reduce method. It connects the variable component of a move operator with a component of the loc-prog that locates a difference to be reduced. It then picks some elements from the domain of the variable and sets up desirable assignments from them. At this point, understanding what it is doing is not as important as watching the transformation that the P undergoes in having its predicates renamed and its contents abstracted slightly.

```
M38 "GEN DES ASG*" (GENRT *DESASG/A.3) (GENRT *DESASG/A.2)
     (COMPON *VARBL/T) (VARBL *LINK/T) (LOC:PROG *LINK/T) (VARBL *DOMAIN/T)
   => (LOC:PROG *LINK/DT) (FILE *DESASG/A) (GENRT *FEASASG/A) (VARBL *ASG/M.1)
     (NOT (GENRT *DESASG/A.3)) (NOT (GENRT *DESASG/A.2))
```

The first abstraction consists of removing the SATISFIES2 and the conjuncts' variable arguments, leaving only primaries and secondaries. The change to the first two conjuncts of the LHS shows how the renaming emphasizes similarity in meaning of predicates, while distinguishing steps in the process. The second line of the LHS shows how interrelations between elements are more transparent. The renaming makes the RHS betray its function much more accurately. The ERASE is replaced by the goal value D, raising what is being erased to top-level status in the conjunct. The "/A" in two conjuncts shows that these are active signals, where before there might have been some doubt, and the use of /M.1 with the VARBL *ASG shows it is an assignment that is only partially specified. Note that the "B" goal value is implicit in the renaming, though in the preferred notation for further work, there would have to be something to occupy each position, in order to make matching reasonable.

A second abstraction can also be obtained, allowing the main function of the P to be seen at a glance. The following has only primaries, with duplicate elements removed.

```
M38 "GEN DES ASG*"  GENRT COMPON VARBL LOC:PROG => LOC:PROG FILE GENRT VARBL
```

Appendix A gives the renamings of the predicates of GPSR. The first half gives the GPSR name followed by the new name, while the second half has the names reversed, ordered according to the new name. From the second half, it can be seen that the number of primaries is relatively small, 29, of which 14 are process primaries, and 15, data. Appendix B gives the first abstraction (as in the above example) for the entire GPSR system, except for task Ps. Appendix C is a cross-reference of that abstraction. Appendix D gives the second abstraction for all the Ps, and in addition divides the PS into modules (to be discussed in the next subsection). Modules are labelled, and are also partitioned, using blank lines, into groups of Ps very similar in form. Given this division, an even more abstract form of the PS can be constructed by merging similar abstract Ps together (this is not shown in the appendices explicitly).

The taxonomy of Section B.2 is closely related to the modifiers of Figure C.1. Evocation corresponds to the "A" process type, iteration to the "G", coordination to the "C", and selection to "S". Cascading is possible because of the existence of degrees and subdegrees, allowing a step to be divided and subdivided as appropriate. Sequencing combines the use of "A" and "C" types of predicates, with the "C" type providing the data for continuing after a step has been completed. With respect to the data aspects of the taxonomy, "I" and "X" indicate inputs to processes (in addition to arguments to "A" items), and "R" and "E" are used for results. Renaming of data to hold it back from being used immediately is done with "H", cleanup is initiated with the "D" goal value, and the desire to evoke knowledge stored as Ps (and in other ways) can be expressed using "K". The goal values add indirection to these meanings: one can "want" to do something, rather than doing it directly, for instance. This allows preparatory activity, application of a method that is essential to applying something else, having "second thoughts", and other similar delaying and interposing. Goal values are not very common in the renaming of GPSR, but are thought to be essential to more demanding understanding-system tasks, where things are expected not to fit together so effectively and directly.

The correspondence of the taxonomy of representation with the taxonomy of control, combined with its effective application to GPSR, supports potential openness. It should be possible to write PSs that can make better use of the PS representation of other processes for their analysis and correction. The structure of names into primary and secondary helps to reduce the total number of names, and might allow the connection of processes associated with a name under one primary to be applied to occurrences elsewhere. A procedure for assimilating information from an external environment or from strange procedures can have an effective means for doing so, requiring only a relatively small amount of knowledge about how things are named. That is, such a procedure can simply do a mapping between representations, with the expectation that if the right names are chosen, some Ps will be able to take processing further.

C.3. Application of the predicate renaming

As a result of renaming, GPSR is divided into 14 processes, corresponding to primaries in the renaming: Eval, the evaluation of goals and differences (17 Ps); Select, which selects old goals and methods (11 Ps); File, which recognizes and canonizes goals, loc-progs, objects, and assignments (52 Ps); Match, which compares objects and extracts differences (21 Ps); Transf, which is the method for transform goals (12 Ps); Reduce, the method for reduce goals (8 Ps); Genrt, which generates move-operator assignments (16 Ps); Apply, which applies or tries to apply move operators (20 Ps); MoveOpr, the method for move-operator application goals (6 Ps); and a set of five operations, add, remove, increment, decrement, and copy (a total of 27 Ps). Each process primary plays a central role in a set of Ps that is the corresponding module. Figure C.2 gives an example of the second abstraction for the Ps in the Transf module (taken from Appendix D). Note the basic similarity of form of the Ps: All except M26 and M27 include a process control signal; M26 and M27 deal with the creation of new subgoals and are keyed to the "W" (want) goal modality. (More detailed versions of the Ps are in Appendix B.)

Using the abstracted forms of the modules, interactions between them can be determined and are of two types: LHS assumptions and RHS actions. Figure C.3 shows

```
M20 " TRANSF G "        TRANSF GOAL => MATCH TRANSF
M20S " SUC TRANS "       TRANSF GOAL => GOAL - TRANSF
M22 " MATCH VAL "        TRANSF EVAL -> TRANSF - EVAL
M23 " MATCH FIN "       TRANSF => MATCH TRANSF
M24 " COMP DIFFIC "      TRANSF => TRANSF GOAL
M24E " ERS MVAL "        TRANSF => - TRANSF
M24F " ERS MVAL- "       TRANSF => - TRANSF
M24N " ERS MVAL SV- "     TRANSF => TRANSF
M24S " ERS MVAL SV "     TRANSF => TRANSF GOAL
M25 " SUC DESCR "       TRANSF GOAL OBJECT => GOAL - TRANSF

M26 " NEW REDUCE "       GOAL => FILE EVAL GOAL
M27 " NEW REDUCE "       GOAL => FILE EVAL GOAL
```

Figure C.2 The Transf module in GPSR

counts of assumptions made in LHSs of the modules' Ps. The counts in the figure are taken from a cross-reference (Appendix C) done on the first abstraction of GPSR (Appendix B), using module boundaries determined by examining the forms of the Ps in the second abstract version (Appendix D). Reading across a row in the figure, there are counts of the number of mentions, in LHSs, of the module at the head of the row. That is, the row counts indicate which other modules are assuming something about the module at the head of the row. Reading down a column, the counts indicate what the module at the top of the column assumes about the others. A vast majority of the assumptions being made are intra-module, with the diagonal of the figure having 82%. Out of 100 entries, 10 on the diagonal are filled, 11 off are filled, and 79 are blank. The order of modules in the figure is based on making as many interactions as possible fall near the diagonal, particularly in spaces adjacent to it. Only 5 entries are outside the near-diagonal region, accounting for about 5% of the total interactions (95% are thus within that region).

	Transf	Match	File	Genrt	Reduce	Apply	Eval	Select	MoveOpr	Oper'ns
Transf	15						3			
Match		14	18							
File			41	1			4			
Genrt			1	16		2				
Reduce					8					
Apply					4	29	1		1	
Eval	1					5	11			
Select								14		
MoveOpr									4	
Oper'ns										38

Figure C.3 LHS interactions between modules in GPSR

The strongest inter-module interaction is between File and Match (without it, 90% of the interactions are intra-module). Recall that in filing objects, the match is invoked and then terminated as soon as a suitable difference is found. To do this, File must know an unusual amount about Match. The full details of the interaction disclose that File uses primarily knowledge about intermediate results in Match, so that it can terminate unnecessary matching by deleting them. This seems to be a weak form of interaction, in contrast to actually assuming how a module works, for instance. Similarly, making a more detailed analysis can "decrease" a few of the other counts, but the changes are not essential to the main claim of modularity.

Figure C.4 shows counts of references made in RHSs of the modules' Ps. Reading across this figure, counts are given for the number of evocations or cancellations by other modules of the module at the row's head. Reading down a column, the counts indicate which modules are evoked or cancelled by the module at the top of the column. This figure omits two counts, due to omission of a column for the task-dependent Ps, which allow GPSR to solve problems. The omitted figures would be a 1 in the Eval row and a 3 in File (filing two objects and the top goal), since the initialization P evokes those two modules to start GPSR running. Task Ps other than the initialization P are included in the Apply module and the Genrt module, and the appropriate values for them have been incorporated in both figures (the task Ps for the Tower of Hanoi are used). The only place where the task Ps make an entry where there would have been space is in the Operations row, Apply column of the RHS table - the task Ps are the only evokers of the operations.

	Transf	Match	File	Genrt	Reduce	Apply	Eval	Select	MoveOpr	Oper'ns
Transf	16	1					1	1		
Match	2	26	9							
File	2	1	63	5		4	2	1	1	
Genrt			1	23	2	1			1	
Reduce					11			1		
Apply				4	9	29	2		4	
Eval	3	1				5	18	1	1	
Select							7	13		
MoveOpr								1	4	
Oper'ns						3				39

Figure C.4 RHS interactions between modules in GPSR

The RHS counts make the system look less modular, but even with the larger number of off-diagonal entries, diagonal entries still are 76% of the total counts. Of 100 possible entries, 10 are on the diagonal, 29 are off, and 61 are left blank. Outside the near-diagonal region, there are 16 entries, accounting for about 10% of the interactions. Note that the "chain" of near-diagonal elements reduces the gain from a decomposition that combines some present modules, since inter-module interactions would still persist. The decrease in modularity doesn't detract from the claim for modularity based on the LHS figure, since evocation is not a strong interaction in the same sense that making use of knowledge about a module in an LHS is. Pure modularity would be 100% on the diagonal in the LHS case, but less than that for the RHS case, since some evocations of other modules are necessary.

Others have studied the problem of measuring modularity and of determining short-term and long-term effects of inter-module interactions. Simon (1969) discusses the behavior of near-decomposable systems. The above tables can be put into the form of near-decomposable matrices by grouping the modules Match, File, and Genrt into a subsystem, and also Reduce, Apply, Eval, and Select into a subsystem. That is, the matrix that results by treating those groupings as single modules is nearly a diagonal matrix, with very few off-diagonal elements. Simon also mentions an order of magnitude difference between inter-module interactions and intra-module interactions as a rough criterion for establishing a clear decomposition. This criterion applies to the module interaction tables with the mentioned groupings. There are more exact mathematical criteria, referred to by Simon, for establishing near-decomposability and concluding that a system will have desirable short-term and long-term behavior, but the applicability of the more exact criteria is unclear at present because it is unclear whether the PSs satisfy the basic assumptions of that formalization of behaving systems.

These figures on modularity suffice to draw the desired conclusions about PSs, answering possible objections raised in Section C.1. It should be possible to build very large PSs without having to impose structural context (subroutining) to reduce global, inter-module interactions. Thus the uniform, single-level property of PSs is likely to be preserved. That Ps seem not to make global assumptions, but rely rather on intra-module interactions, makes their incremental augmentation and modification tractable, and adds to the possibility of having their knowledge be effectively provable.

D. The Nature of AI Programming

D.1. Themes of control

This subsection will discuss major control themes, with two purposes in mind: first, to isolate the essential features of the tasks done in this thesis, with a view toward improving the set of benchmarks; second, to explore how various underlying architectures have an effect on the implementation of these themes, using PSs as an example. The themes are put forth as being characteristic of a broad range of AI programs. Primarily, we aim to set up a basis for general evaluation of present and proposed AI architectures. One means to this is establishing a set of benchmark programs, whose implementation reveals central features and provides convincing evidence that there is adequacy for a much larger set of AI systems. A set of benchmarks should thus span a wide range of capabilities, but should try to avoid redundancy of mechanisms so that as much as possible can be gained from each implementation. The discussion here of the themes that make up the PSs implemented helps bring out the benchmarks' structure and raises considerations that may lead to other evaluative approaches.

The following themes require organizational and control facilities that are more demanding than the control features given as basic in Section B.2 (iteration, selection, etc.). They are more demanding at least in the sense of requiring a combination of several techniques. These themes are present, for the most part, in the completed PSs.

And-Or goal sequencing, with recursively-nested goal structure;
Backtracking and other forms of extended iteration and generation of possibilities;
GPS-like heuristic search executive, involving general method-coordination, evaluation of progress, and allocation of effort;
GPS match, involving the extraction, cataloging, and evaluation of differences between complex, structured problem states;
Data-directed or pattern-directed problem-solving strategy;
Natural language processing, including disambiguation and coordination with the pragmatics of the domain under discussion;
Automatic acquisition of new knowledge, both procedures and data.

A number of indications of the power of PSs follow from the implementations of these themes. That is, the themes are achieved, in many cases, with unexpected ease, avoiding a number of traditional mechanisms. A number of the programs exhibit complexity without the conventional structural hierarchy programming style. Hierarchy and other structuring is achieved rather easily using the seemingly weaker programming facilities provided by the PS architecture. Using PSs allows an approach to natural language processing that avoids a conventional syntactic parsing mechanism. Backtracking is achievable in PSs, also without the kinds of control primitives specifically added to other AI languages for that facility. In PSs, structures that are learned by a program need not be interpreted by some part of the program, but can be encoded as active structures, behaving in ways similar to the rest of the system. The global Working Memory is crucial to this capability, in allowing a program to effectively monitor the action of such newly-added pieces of knowledge. From this, it is clear that useful insights can be gained about an architecture by implementing programs that include the above themes. The insights about PSs given here were by no means predictable in advance, and in many cases emerged only as the language and its capabilities were exercised in actual programs.

There are other effects of an architecture on implementations, though these are not nearly as evident from gross program features, or as easily measurable, as is the shape of how the themes are produced. The following useful features of PSs are potentialities suggested by actual practice, and exploitable by further research, rather than features demonstrable by implementing the PSs to specific levels of performance as required by the benchmark concept. A major proposal for an AI architecture might be given an initial evaluation by trying to achieve a corresponding set of functional capabilities.

Global Working Memory for general communications.

Architectural flexibility, deriving from degrees of generality and specialization that Ps can have in practice and from the alternative memory structures available; the effect is to allow plenty of room for design.

Small size of Ps and at the same time the amount of action accomplished by a few elements; this allows PSs to be programmed incrementally, and potentially automatically.

Independence of Ps within the P Memory, and the lack of structure of P Memory (for instance, as subroutines); PSs are open for combination into larger systems, with ties between such program fragments provided by similarly open Ps, through the global Working Memory.

Abstracted Ps as a feasible way of describing the action of a PS; abstract Ps (APs) and very abstract Ps (VAPs), which are used throughout the thesis to represent PSs for descriptiv purposes, retain the style of the more concrete PSs and indica a unifying organizational framework at all levels of abstraction, certainly a rarity among programming architectures for AI.

The present set of programs is useful for benchmark comparisons, with two possible exceptions. The use of Epam as a task is probably redundant. It seems feasible that the mechanisms within GPSR are sufficient to perform the Epam task, and in fact GPSR includes several kinds of Epam-like networks, with the object network actually an improvement over the design used in the Epam PS. Complete details of doing the Epam task in GPSR have not yet been worked out (and space would not permit it here), but the main idea is to extend GPSR slightly to allow it to add operators during the problem-solving process, and then to give it the task of building a set of operators to produce a list of syllables. This could easily be done, by my estimate, with fewer additional Ps than the 41 that constitute the Epam PS. The second possible exception to usefulness as a benchmark task is the restricted chess task. The primary themes addressed by that task are the use of a heuristic search executive and the potential for data-directed problem-solving. The former is central to two other tasks, GPSR and WBlox, and the latter might be just as effectively explored by elaborating the task given to the blocks problem-solver, for instance. On the other hand, the evaluator of a new AI language might want to emphasize performance aspects that are best exercised by a task like chess, where potentially a large amount of search is done. That is, emphasis in a language might be on making search most effective. Decisions on such borderline cases are best made in connection with particular studies.

The set of benchmarks must be augmented to meet the demands of growth of the AI field. The following themes are proposed to make the set of benchmarks more complete, but most of them are not considered necessary to pursue in order to be confident about the applicability of PSs to original areas of research (see Section E.3). Each theme is accompanied by suggested complete programs that include it.

Best-first heuristic search, with problem states or contexts too large to maintain as distinct objects, as in GPSR; GPSR uses primarily a depth-first search, but could be easily modified to be more along the best-first lines (some suggestions in Chapter IV elaborate on this); an essential part of a task to exercise this would be flexible use of information from a number of distinct contexts, and flexible switching of effort from one to another; some of the more advanced blocks problem-solving systems are of this nature (Fahlman, 1974, and Sussman, 1975).

Semantic network or a similar knowledge structure, involving combinatorial search through relational structures and operations on knowledge such as mapping (Moore and Newell, 1973; Shapiro, 1971);

Extension of the natural language task to one of comprehension of larger units (Schank, et. al., 1975);

Search among competing hypotheses using diverse knowledge sources (Erman and Lesser, 1975);

Induction of patterns from examplars (Winston, 1975, Hedrick, 1974); the Hedrick formulation of the task has the advantage of making use of a semantic network, thus combining themes;

Automated design (Eastman, 1973); this task involves use of information in fulfilling vaguely-stated objectives and perhaps trading off various objectives, rather than problem-solving toward a definite goal; a task like blocks manipulation might be extended to include this theme.

A current task form is the construction of understanding systems themselves, rather than attacking singly the variety of themes that seem to be requisite for this larger aim. Perhaps a domain will be discovered that combines the themes in such a form as to be an effective benchmark. Benchmarks, however, must also not involve too much domain knowledge, so that more complex understanding systems are automatically ruled out.

In conclusion, although these themes and the discussion here indicate something of the nature of AI programming, the sharpness of the distinctions between architectures is not fully satisfactory. That is, the form of the PS architecture is reflected to some extent in the form of the above themes as they appear in the PSs, but looking at the themes alone is not sufficient to determine all features of an architecture, for two reasons. First, it doesn't bring out the same kind of information as is brought out by the more detailed analysis that arose from the features of Section B.1 and their application to the taxonomy of Section B.2. Second, it doesn't exercise enough those capabilities that are new in the architecture, in a sense only exposing the potentials for significant advance. Perhaps some systematic way of forcing unexpected augmentations to a system after completion to the predefined benchmark, could bring out more of the dynamic potential of an architecture, as opposed to simply testing its feasibility in a circumscribed task.

One might also question the entire benchmark concept, and try to examine the present results from the standpoint of avoiding the programming altogether. Perhaps an analytic technique could be based on the features put forward in Section B. But some aspects of architectures seem evident only in building complete systems, and in bringing them to some predefined performance level. More analysis might reduce the number of actual systems built, by forming tasks that include more of the central themes. We might even hope for a single, comprehensive task, but this must wait until we can better characterize the essential nature of AI. In any such attempt it must be clear that the desired themes are being exercised in all of the important ways, and especially in those ways that are somehow critical in discriminating architectures. How to formulate the construction and evaluation of critical test cases will persist as an important research area.

D.2. Problem spaces as a basis for a theory of AI programming

The concept of problem space is central to the analysis of human problem solving behavior put forward by Newell and Simon (1972, p. 59, pp. 810-811). A problem space is a means of expressing the possibilities for behavior, rather than restricting a description to the actual behavior observed. It thus describes more completely the problem solver, and even provides a basis for prediction. It has five components: (1) a set of elements, each representing a state of knowledge about a task; (2) a set of operators that produce new elements from existing ones; (3) the initial element, the state of knowledge at the beginning of a task; (4) the desired element, or set of elements, whose attainment constitutes a completion of the task - attainment achieved by applying operators to elements starting with the initial one; (5) and the total knowledge available, ranging from temporary dynamic information to long-term reference information.

The most useful form for a theory of AI programming would be one that would provide an initial framework with which to start the process of programming. That is, one would want something that would apply to an initial statement of a problem and immediately organize it so that the succeeding steps of filling in more detail and encoding it in some language (preferably PSs) would be streamlined. What this subsection attempts to do is to motivate the use of problem space by pointing out how the PSs developed for this thesis can be formulated as problem spaces, and also by pointing out how well-suited PSs are for exploiting the structure imposed on a problem by the problem space framework. Three assumptions are made in carrying this forward: first, that a problem space framework is relatively easy to develop for typical AI programs, when one is in the initial exploration stages; second, that the correspondence of the final form of the completed PSs to problem spaces means that there can be something like a problem space framework guiding the process of constructing a PS from the beginning; and third, that the representation as a problem space means that processes of programming can take advantage of it in those problem space terms, with the problem space structure clear enough to use. The first assumption may be made more plausible by illustrating the application of problem space framework to the programs of the thesis, but it cannot be shown valid without taking some new problem and attempting the same application process. The second assumption is even more difficult to support, seeming to require at least detailed study of the incremental construction of PSs. The third assumption is an instance of a more general principle, that representation affects processing done on it.

Problem spaces structure a task by dividing applicable knowledge into a relatively small number of operators. Each operator (by definition) can take action in a number of ways, and it is this variability that allows a small set of operators to generate large behavior spaces. That is, if we take each knowledge state (problem-solving dynamic state) as a node in a graph representing the search space, then an operator applied to a knowledge state can potentially cause the graph to branch out in a number of directions, each representing the transition to a new, distinct knowledge state. How it branches in a particular case depends on the content of the state.

As we have seen above, as a result of the renaming of predicates, a PS is divided into a relatively small number of main processes, each composed of between roughly 10 and 50 Ps. What better representation than as a set of Ps could there be, for the kind of variability inherent in problem space operators? PSs have two distinct advantages: The behavior of PSs divides into relatively small sequences of unconditional actions, corresponding to Ps, so that there is a high degree of conditionality and so that Ps can be seen as units of variabilty. PSs act by global communication, with potential access to a full knowledge state and action on a full knowledge state. For the moment, this suitability about PSs is hypothetical, of course, and it needs to be qualified by saying that although the most natural correspondence would be to have each possible full action of an operator represented by a single P, in practice it must be allowed that a sequence of P firings within an operator is necessary to develop its action. In fact, in applying the problem space concept to the Studnt PS, each operator averaged around 15 P firings. When such a number of Ps participates in determining the action of an operator, it must be the case that the amount of variability in the resulting knowledge states is correspondingly large.

Another possible correspondence between PSs and problem spaces is that problem space operators are coordinated on a large scale in a fashion similar to the way Ps are coordinated on a small scale. A problem space operator is somehow matched to a knowledge state to produce a new one. The result is a relatively small amount of action on a global set of knowledge states: the addition of a new one. Most importantly, the openness of the selection of an operator to apply to a state corresponds to the openness of selecting a P for firing: problem space operators are generally not described as participating in some sequential procedural framework, but are stated more as data-directed, relatively independent entities. This is not completely true for all problem spaces, because there exist in some spaces sequential plans. These plans serve to tie together the application of several operators into a coherent sequence. Plans can be temporary, task-dependent shortcuts to solutions, or they can be used effectively in many situations, in which case they become a form of stereotyped behavior and move away from being part of problem solving behavior. But this is analogous to sequences of Ps that become sufficiently common and useful to be convertible into a single P with a longer unconditional action sequence.

Before going into detail on the particular PSs seen as problem spaces, there is a qualification to our adherence to a strict definition of problem space. The concept of problem space is being discussed here at a very abstract level, with the consequence that it is in some cases an idealization of what the essence of problem space is. The details of the definition of problem space given by Newell and Simon have been modified somewhat to apply to the broader domain here. On the one hand, we are not concerned so much with the detailed theoretical implications of the definition of problem space for cognitive psychology. On the other hand, the number of examples that were explored in the problem space framework in the original defining work is so small that some distortion and modification is almost inevitable.

The Studnt PS has as its task to convert a string of words into a set of algebraic equations and a specification that certain variables of those equations, the unknowns, are to be solved for. A knowledge state for Studnt consists of a partially-scanned string of words, along with internal symbol structures that represent the status of the process of constructing the equations and unknown variables. Speaking broadly, the problem space operators apply to such states to produce increments of progress, represented in new states that have less unscanned string or less internal partial structures, and more of the final result. The space of possibilities is large because of the astronomical numbers of equations that can be formed from grammatical strings of even relatively small size.

The problem space operators are divided conceptually into three sets: the initial scan operators, the parsing operators, and the operators for segmenting unknowns. The initial scan operators are of three types: a transformation operator, which replaces idioms in the problem string by other standard forms; a dictionary-tagging operator, which classifies certain key words, for the use of later operators; and an initial chunking operator, which forms the main sentential chunks from the string and notes their main connectives (which are "is" or some arithmetic operator). In applying these initial scan operators, the Studnt PS makes use of a plan that resolves certain ambiguities with respect to which of them might apply to the problem string by invoking them in a particular order and also according to a strict left to right scan across the input string. The initial scan operators work directly with the input string, producing a modified string and ultimately converting that string into a chunk, which is a string that has specific boundaries, a unique name, and other properties. Chunks are the primary components of the internal symbol structures that combine with the partially-scanned input string to make up knowledge states in the problem space.

The second set of operators are for parsing a chunk into an equation: one operator scans a chunk to find an appropriate place to split it into component chunks; a second operator identifies a chunk as a variable, as not subject to further subdivision, and also checks whether that variable is the same as a previously identified variable chunk; and a third operator recombines variable chunks into expressions, which can then be taken into more complex expressions by further combinations, using information associated with the chunks when they are split by the first parsing operator. The parsing operators thus take chunks from a knowledge state and operate on them to produce further chunks and also expressions, which are closely associated with chunks, rather like their other attributes. An equation is a particular kind of expression.

The third set of operators has only one element, a special operator for splitting a chunk recognized to contain the specification of the variable unknowns into the appropriate unknown chunks. When those chunks are determined, they are identified with previous problem variables by the same procedure that is applied within the parsing operator. The parsing and unknown segmentation operators are organized into plans in ways similar to the organization of the initial scan operators, and for similar reasons.

GPSR aims to find a sequence of task operators that apply successively to an initial symbolic configuration to produce a desired configuration. To do this, it sets up an internal knowledge state organized around goals. Thus to GPSR, the space being searched is a space of goal trees, and only secondarily a space of task configurations. That is, the space GPSR is searching in is not the space of task operator sequences, such as the Move-Disk operator in the Tower of Hanoi problem or the Cross-River operator in the Missionaries and Cannibals. It is something more: task configurations are incorporated into a richer description space built around goals to transform task configurations from one to another, goals to reduce differences between task descriptions, and goals to apply particular (partially-instantiated) task operators. GPSR's knowledge states, in addition to containing a general goal graph structure, are composed of a current status for the problem-solving executive and a number of auxiliary task-dependent structures.

The process primaries derived from the renaming process in Section C are candidates for problem space operators: Apply, Evaluate, File, Generate, Match, Move-Operator, Reduce, Select, Transform, and the Operations. We can narrow down this set by taking the problem space operators only those members that involve significant problem solving, i.e., that represent places where a number of possibilities exist and where the operator goes with one in preference to the others. Evaluate takes a goal, either new, old, newly-succeeded, or newly-failed, and produces an evaluation of it, with the result of augmenting the current knowledge state by making some goal (either the goal immediately input to be evaluated, or another goal selected according to its evaluation) the current goal. For our purposes, it can be said to include the File and Select processes, since they don't make changes to major components of knowledge states and since they are dynamically subordinate. Select does, however, do significant problem solving, so that it could be seen as an operator closely linked by a plan to the Evaluate operator. (Some versions of GPS include goals of select type, in which case Select is augmented beyond its GPSR form and is more independent of the executive.)

The Transform process is capable of recognizing when a solution is attained, and otherwise is the evoker of Match, which results in establishing a new reduce goal. Match is subordinate to Transform and also to Evaluate, and is not considered to be an operator, since it is subordinate and since it doesn't do problem-solving in the sense of selecting from alternatives in the space. Instead it produces an exhaustive list of differences and leaves the evaluation and selection of those to its parent processes. (Even in augmented forms of Match in GPS, where so-called immediate operators are added, there is no problem-solving in the present sense because the immediate operators are necessary for the match to proceed.)

Reduce is closely tied by a problem space plan to two other operators, Generate and Apply, which do a significant amount of problem-solving. Reduce takes the focus on a reduce goal and selects a task operator to be applied to reduce the difference attribute of the goal. It then evokes Generate, which connects information about the task operator and the difference to be reduced to form a set of desirable assignments for the variables of the task operator. Generate also extends these assignments to full, feasible assignments. The Reduce plan then calls for the evocation of Apply, which takes the set of feasible assignments and checks the result of applying task operators for each. Apply selects from the results of the application attempt, to produce a success signal (a modification of the executive status for a goal) or information that is used to construct a new goal.

The Move-Operator process is little more than a plan to evoke the Apply operator to try to apply a task operator, and then to construct a new goal according to the result of Apply. It doesn't do much problem-solving, but it has a visible effect on the knowledge state, so it deserves operator status. The Operations processes are subordinate to Apply and do only straightforward symbolic manipulations to attributes of goals, so they are not problem space operators. Thus, GPSR includes the following problem space operators: Evaluate, Transform, Reduce, Generate, Apply, and Move-Operator. This discussion of GPSR is important in illustrating the use of the structure evident from the renaming of Section C. That GPSR's process structure and its structure as a problem space correspond so closely is strong evidence for the argument that problem spaces are effective as a theory.

The two examples already given should give a good idea of the form of the argument for problem spaces. Problem spaces for the other four PSs need only be briefly sketched in order to provide additional support and completeness of coverage. The task for MiliPS is to form a unique interpretation of an input string, maintaining both consistency with a model of a toy blocks scene and naturalness of the interpretation (as opposed to finding an interpretation that would not occur to a human in a similar situation). In some cases, MiliPS recognizes some kind of error in the input, and provides a diagnosis of the problem as its output, e.g., by describing specifically the sort of ambiguity detected. A knowledge state has the remaining unscanned input string, a list of objects encountered in the sentence that can still be useful for making further interpretations, the unresolved ambiguities in sentence, and unused relations and other structural fragments that are to be filled in by more scanning of the input. MiliPS searches in the space of possible interpretations by applying operators to lexically classify words, to verify grammar, to create and identify objects associated with nouns, to apply attribute values and relations to restrict ambiguities, to resolve inconsistencies and redundancies, to describe scene objects, and to deduce and perform the actions that are the main intent of an input. These operators are all represented by sets of Ps in MiliPS, and are easily distinguishable as program units. Some of the mentioned operators do less in the way of reducing the remaining space of possibilities for an input, i.e., do less problem-solving, than others, with the reduction of ambiguities, inconsistencies, and redundancies estimated to be the most important.

The space of possibilities for Epam is the space of extensions of an existing discrimination network in order to improve performance on the syllable task. In a sense, it is searching a space of networks. Primarily, Epam consists of a primitive matching process, which compares its behavior to the desired behavior, and an extend-net operator, which takes action on the diagnosis of a difficulty produced by the match. In contrast to GPSR's match, the match in Epam does do some problem-solving, distinguishing between various cases to be corrected. The extend-net operator, however, is responsible for the majority of the problem-solving.

WBlox consists of a number of task operators for manipulating blocks worlds, only some of which are problem space operators. It is given a particular blocks configuration and a command to be executed on that configuration, with the command amounting essentially to a partial description of a desired state. Since there is little in the way of internal goal descriptions (in contrast to GPSR), the knowledge states are taken to be basically blocks configurations, among which WBlox searches with a variety of operators in order to achieve the desired one. The operators chosen for the designation of problem space operator are the intermediate-level blocks operators, as opposed to the lower-level commands, which do no problem-solving and are subordinate, and the high-level commands that initiate the system's activity but that don't do anything directly. Thus the problem space operators are the following, which do produce different configurations of blocks when applied: PUTONSET, which is an iteration of the basic PUTON1 operation, but with the capability of trying alternatives; STACKUPSET, an iteration of the PUTON1 operator, similar to PUTONSET in problem-solving capability; PUTON1, the placement of one block somewhere on top of another; PACK, an iteration of PUTON1 and other more primitive operators; FINDSPACE, which finds space to put something, doing a small amount of problem-solving to arrive at a suitable location; GRASP, which should really be called TRYGRASP, an attempt to grasp an object that may involve some rearrangements before being achieved; GETRIDOF and CLEAROFF, which also do rearrangements to place objects in non-interfering locations; and MAKESPACE, which rearranges blocks to force the availability of open space.

Finally, KPKEG searches among chess positions, with the basic knowledge states augmented by information about strategies being tried and alternatives still available. The primary operators are an evaluation operator, similar to GPSR's executive, and a generator of moves that fulfill strategic objectives established by the executive. If one wanted to refine the generate operator, it could be broken down into a small number of strategy-specific generators.

The above presentation of problem spaces has emphasized the operators at the expense of describing details of knowledge elements and total knowledge available. This is because the operators are the generators of the behavior spaces, and are the most visible components in the PSs, since each operator is a set of Ps. It should be pointed out that often the actions of problem space operators have been described abstractly in the body of the thesis as very abstract Ps (VAPs). But it is also the case that in doing the above descriptions, there is ample contact with general task concepts, and the emphasis is not entirely on the opposite bottom-up considerations, the way that the PSs, which are only particular implementations of the tasks, correspond in organization and in detail with problem spaces. In some cases, the presence of problem-solving within operators has been portrayed as central, since it is through the existence of problem solving that there is a potential for a space of possibilities. That is, problem solving is seen as the application of knowledge to make decisions of some sort, and it is the possibility of making decisions in a number of ways that makes the space.

In conclusion, there are a few points to be made on the apparent advantages of implementing problem spaces with PSs. The taxonomy of control presented in Section B can be seen as a kit of techniques for implementing problem space operators. The level of the various elements of the taxonomy is such that it doesn't cross the conceptual boundary of a problem space operator. That is, the control techniques are right for doing operations within problem space operators. Also, nothing in the taxonomy proposes any overarching organization that would conflict with the problem space view. When implemented as PSs, problem space operators become rather open in terms of inter-operator interactions, both in terms of the size of action done by a P (interruptability) and in terms of the globalness of all Working Memory interactions. That the inter-operator interaction doesn't get out of hand is demonstrated by the modularity measures in Section C. Finally, augmentation of a PS program can be viewed in two ways: the augmentation of the operator set, which should be facilitated by modularity and globalness of interaction and evocation of all the operators; and augmentation of particular operators by adding Ps to represent further behavioral possibilities, also aided by precisely the same factors.

Having verified to some extent that PSs are right for problem spaces, we still have to examine the larger question of how fruitful it is to view AI programs as utilizing problem spaces. Certainly the view is supported by the specific correspondences for the six PSs above. But perhaps we should reconsider the definition of problem space and note that it includes some amount of search as contrasted in the extreme case with an algorithm that performs directly without such intermediaries as goals and subgoals (Newell and Simon, 1972, pp. 820-823). Thus our assertions about the nature of AI programming and the suitability of PSs should be applied only to research of an exploratory form, a form that is common in most past AI research and that seems inevitable in dynamic, open-ended understanding systems.

E. The Future of Production Systems

E.1. Serious defects

Run-time efficiency is the primary weakness of the PSs implemented here. To summarize a number of comments made in connection with particular PSs, run-time is too large for practical purposes by roughly an order of magnitude (more precisely, a factor of 6 to 10 times). One effect of this is that the programs are too slow to be run interactively, and in practice, much of the debugging for the thesis was done in batch mode, with only one or two runs per day. At present, this is diagnosed to be due to correctable causes. The Psnlst interpreter has several known, low-level inefficiencies, and probably more, simply because not enough time was taken to make the internal algorithms and data representations more optimal. As has been discussed in Section B.5, there is no clear evidence that there are costly factors inherent in the PS architecture (such as increasing expense as the number of Ps increases), except perhaps for the presence in Working Memory of a large amount of useless information that has not been properly cleaned up. This is also in agreement with another study of PS efficiency (McDermott et. al., 1976). But most importantly, PSs are amenable to significant efficiencies beyond improving isolated architectural features and interpreter implementations: the possibility of representing PSs in a compiled form. This involves transforming the external representation of the Ps into a form that takes account of repetitive and redundant matching operations, converting them to a form optimal for the matching algorithms and making provision for the storage of partial results to avoid duplications. Forgy (1976) and McDermott et al. (1976) describe some initial efforts in this direction. Forgy says that PS efficiency might be improved on by at least a factor of 5 (over the present Psnlst), and perhaps more, by proper compilation, and also that properly-designed hardware could achieve quite a bit more than that, up to more than 15 times the present Psnlst. It is expected that efficiency will be a top-priority item for PS research, but also that its resolution will be relatively rapid.

The other major deficiency with the PS implementations here is the ad hoc quality of the control and data representation. The proposed shift in representation outlined in Section C.2 is a response to this feature and all its consequences. A related feature is the excessive use of control as opposed to letting more processing be more open, specifically data-driven or bottom up. The tasks chosen did not test the architecture along this attribute, as was implied in the discussion in Section D.1 and below in Section E.3. There can be no doubt that PSs are well-suited to such a style, and it is likely that this capability of PSs will be exercised more when PSs are applied to tasks that also make use of the openness resulting from the proposed representation shift.

E.2. Promising features

A number of features of PSs are indicated by the programs done, but the tasks were not carried far enough to allow them to be actually demonstrated. A cluster of capabilities revolves around the potential for automatic creation and modification of Ps, where automatic refers not so much to deliberate actions within PSs but to other more general processes that can apply in an unrestricted way to running PSs. First, there is the possibility of collapsing dynamically-adjacent Ps into shorter, more task-specific ones, which could accomplish some action with fewer recognize-act cycles in particular cases. This was emphasized most in connection to GPSR, where a number of "interpretive" aspects of the processes were amenable to being tuned to take advantage of fixed properties of the various tasks. There is the possibility of taking more advantage of the converse of collapsing sequences of firings, namely to break down unnecessarily complex Ps into simpler combinations or cascades of Ps, with the benefit that the result is more general, capturing a number of cases for which there weren't complex Ps before. This was noted in connection with GPSR and KPKEG. If, for instance, a set of Ps test the interaction of a number of factors, say 6, divided into two sets of 3 related tests, then to have single Ps perform all the possible combinations would require 9 tests, whereas 6 Ps suffice if the test is broken down into a sequence of two tests requiring 3 Ps each. This mechanism especially pays off when the smaller Ps are accidentally applicable to situations outside the cases to which they were closely tied (as distinguished from simply filling in missing cases within the local combinatorial ones). The utility of each of these two P-modification operations is that they could be tied to the frequency of usage of Ps, so that such "optimization" would be applied only where suggested by tasks. All this is not to say that such automatic augmentation processes are significant enough to become an overshadowing factor in the power of PSs, but that they might operate in the background to improve PS capabilities and exploit inherent architectural flexibility.

Augmentation of PSs could be made automatic by periodically forming new Ps from changes occurring in Working Memory. That is, a condition is formed from some set of older Working Memory elements, and an action, from some more recent set, the two sets thus associated together as a P simply by time adjacency. This was discussed as an a priori property of PSs in Chapter I, and has yet to be explored except in prototype studies. Similar Ps formed in such fashion could conceivably be collapsed into smaller numbers of more general Ps, simply by converting selected constants to variables ("selected" referring to constants that differ or clash in otherwise similar parts of Ps).

The use of the representational taxonomy of Section C.2 is promising from the standpoint of combining separately-developed systems to obtain new interactions beyond a simple sum of their properties. This would certainly be facilitated by keeping the number of primary processes as small as possible, and by keeping them open for application or mapping to new task areas. Structural features of GPSR (see Chapter IV), for instance, make it open for use as a module in other systems.

Finally, there are ways that PSs lend themselves to more power, both in general and in specialized task domains. The recognize-act cycle might be modified to allow more of the multiple firings, along the lines of the ones that occur now when there are a number of possible matches to the same P. One possibility is to allow a number of different Ps all to fire on the same cycle, when they are true and at the same time are keyed to the same event or change to Working Memory. This would increase the power of PSs to do iteration and to express essentially asynchronous processes, decreasing the need for deliberate control mechanisms. Specialized power follows from the PS architecture when the language is modified to take advantage of peculiarities of tasks. This possibility comes up in connection with chess, where the central task representation, the chess position, is something that is common to much processing and could be streamlined to be expressed in the language and in the underlying implementation more efficiently.

E.3. Gaps in the evidence on production systems

One major area of AI programming untouched by the PSs of this thesis deserves some discussion here: the area of encoding knowledge in the form of semantic networks. A useful formulation of this task will include two aspects of understanding systems: operability of knowledge and automatic encoding of knowledge. The PS approach to semantic networks will be sketched briefly below. Some important aspects of operability of knowledge are mapping or conversion of Working Memory elements so that existing Ps can be applied to them (termed assimilation by Moore and Newell, 1973); formation of new concepts; formation of problem spaces, as discussed in connection with the Studnt PS (Rychener, 1975); and the modification and augmentation of existing knowledge embodied in processes and about processes. The PSs developed here don't make much contact with a number of other systems that use Ps in a radically different way, RHS-driven, "goal-oriented" production-based systems (Davis, Buchanan, and Shortliffe, 1976). Such systems have achieved a moderate amount of operability. My current thinking on this other form is that it may be a more primitive form, and that a transitional sequence might be found to connect the RHS-driven form, by a series of collapsing and aggregating operations, with the form common to this thesis and a number of others (see Chapter I for related work).

The PS approach to semantic networks follows the same principles used for discrimination networks in Chapters III and IV.⊛ Semantic connections will be represented by Ps, rather than by relational structures in Working Memory interpreted by Ps. The firing of a P will represent the traversal of an arc (or arcs) in the conventional network, and that firing will result in leaving in Working Memory a temporary state, the internal state of the network executive or searcher, as it were. This is not unlike the process described by Rumelhart et. al. (1972).

Querying the information in the network would be by constructing a P or set of Ps that would monitor the changing Working Memory state and fire on recognition of an answer, either positive or negative or something else, depending on the stringency of the test. Multiple-origin searches could be carried out by firing Ps in parallel, especially effective if the conflict resolution is loosened up to allow several different Ps to fire at once, as suggested above. Figure E.1 gives a fragment of a network, using simplified Ps.

The N Ps, the network proper, are two simplified classification hierarchies, one for "tulip" and one for "dog". The first three Q Ps represent three questions that might be posed to such a network: "is Dog-7 an animal", "is Tulip-3 a plant", and "is Tulip-3 a dog". The Q4 P represents a general piece of information, "plants can never be animals", the sort of thing that would be used to answer the third question in the negative. For each of the questions, a search involving firing of several Ps takes place, with one of the Qs ultimately providing an answer. For instance, to answer the first question, "Dog-7" would be asserted, resulting in firing N1 (producing isa-dog(Dog-7)), then N3 (isa-canine(Dog-7)), then N4 (isa-mammal(Dog-7)), then N5 (is-animal(Dog-7)), and finally Q1 (answer(yes)).

⊛ The approach was developed in conversations with A. Newell and D. Waterman.

```
N1: Dog-7 -> isa-dog(Dog-7);
N2: Tulip-3 -> isa-tulip(Tulip-3);
N3: isa-dog(x) -> isa-canine(x);
N4: isa-canine(x) -> isa-mammal(x);
N5: isa-mammal(x) -> isa-animal(x);
N6: isa-tulip(x) -> isa-flower(x);
N7: isa-flower(x) -> isa-plant(x);

Q1: isa-animal(Dog-7) -> answer(yes);
Q2: isa-plant(Tulip-3) -> answer(yes);
Q3: isa-dog(Tulip-3) -> answer(yes);
Q4: isa-animal(x) & isa-plant(x) -> answer(no);
```

Figure E.1 A fragment of a semantic network

Note that in the case of the third question, two searches need to be done, namely one starting with Tulip-3 and one starting with the assumption that "isa-dog(Tulip-3)", with the searches ultimately producing the contradiction recognized by Q4.

The Ps above are simplified. An actual system needs some guidance of the search, and it needs some way of stopping the search. Guidance can be provided by adding extra conditions and actions to the network Ps, and by adding extra Ps to monitor the state of the search in Working Memory, performing deletions to prune the search. Stopping the search might be achieved by having extra Working Memory elements record a search activation level, updated each time a network P fires or each time some recognizable event occurs.

From this preliminary presentation, PSs can be seen to have several positive features for this task. Specific heuristic information can be encoded directly in the network, not, for instance, in some all-knowing centralized executive, where its access might become rather involved. The network is actively encoded, and takes advantage of the power of PSs to perform the iteration of the search cycle. And the patterns searched for are limited only by the expressive power of Ps. This means that there is no limit to the diversity of knowledge brought together for an answer, or to the processing done at each step in a search. Enthusiasm for this approach must, however, be tempered by the obvious difficulty of overall control: with so many Ps firing in asynchronous fashion, there may be a requirement for a large number of general querying and (domain-specific) search-limiting Ps to contain the search. No task analyses exist (to my knowledge) that would provide data on the potential difficulties here. For instance, there are no measures on the "branching factor" of semantic nets, and in particular there are no comparisons of branching for networks used in various tasks.

E.4. Practical, impractical, and theoretical applications

PSs seem ideally suited for high-level cognitive functions where large amounts of domain knowledge are to be brought to bear. Such tasks exploit the understanding system properties of PSs, particularly the way that knowledge is added incrementally and interactively. Some specific applications are: knowledge-based systems, where an expert's knowledge is encoded and applied to real problems; computer-aided instruction, a field with unlimited room for expansion of the knowledge base, while at the same time demanding a concurrent augmentation of processing capabilities; human-computer interfaces in general (particularly natural language ones), where the interface must be adaptable and perhaps even capable of modelling its users; tasks familiar to cognitive psychologists such as protocol analysis and modelling; and cognitive components of robotics, speech understanding, and visual systems. These areas are contrasted to low-level number-crunching applications, such as acoustic and visual-image preprocessors. PSs might not be best in areas where little is known and reliance must be primarily on brute-force exhaustive search techniques. But two qualifications can be made to these apparent misapplications of PSs. First, even in exploratory studies, the incremental property of PSs might allow effective and rapid narrowing of a large search space. Second, since PS interpreters can be coded relatively easily, there may be cost-effective specialized PSs for, e.g., tasks involving numerical computations that are somewhat condition-dependent.

On the theoretical side, PSs are well-suited to several kinds of exploration. Generally, PSs are a transparent medium for exploring the content and form of knowledge in a domain. PSs are good for exploring new ideas, given the rapidity with which a working system can be constructed. Much theoretical, application-independent work remains to be done in the area of automatic encoding of knowledge, i.e., building instructable PSs. PSs have proven effective in this thesis for replicating past AI efforts and in more detailed analysis of past work, and are a concise means of expressing programs for documentary and descriptive purposes. Using PSs, the field of AI might yield to analysis aiming toward a comprehensive rationalization or systematization. Finally, PSs might prove to be useful to current explorations of parallel computer architectures, as a simple computational mechanism that allows complex systems to be broken down into a number of asynchronous modules, or perhaps as an abstract formalism for such systems regardless of the actual implementation.

E.5. The case for production systems

PSs are effective and advantageous for the programming constructs typical of AI systems. The six PSs implement systems with a variety of methods and representations. Of nine programming language properties discussed, PSs have particular advantages in style, conciseness, and architectural flexibility. They also are favorable with respect to practical feasibility, productivity, and degree of being guided by a theory. Their attributes are mixed on power and overhead features, and are negative at the moment on efficiency. Some of these evaluations are strictly comparative, while others can not be comparative at present due to lack of similar measures for other systems. Mechanisms of PS control are encompassed by a relatively concise taxonomy of six process-evocation categories and five data-management categories. Major successes can be expected in applying PSs to large-scale understanding systems of the sort currently being explored. Of a set of seven secondary understanding system properties, four are supported by a priori PS properties and are further supported by the six PSs. Two other properties, openness and modularity, are supported by the application of a taxonomy of representation, and the seventh, provability, has not been attacked by the present methods. Two primary understanding system properties, operability of knowledge and automatability of the encoding of knowledge, have not proven amenable to demonstration by the present approach, and are left open for further research.

PSs are particularly useful in domains where system knowledge must grow dynamically through interaction with humans and with a task environment, but without the expense of analysis of how each new piece of knowledge must fit into existing structure. A set of major themes of control in the systems implemented stand as hallmarks of AI programming and may prove useful in evaluating new and proposed system architectures for AI. A preliminary theory of AI programming can be based on the correspondence of the PSs with the concept of problem space. Such a theory may provide a framework for the organization of future understanding systems, especially given PS properties. Diversity of application and problem-solving capabilities, both of which are deemed essential to building understanding systems, have been adequately demonstrated.

F. References

Davis, R., Buchanan, B. and Shortliffe, E., 1975. "Production rules as a representation for a knowledge-based consultation program", Report STAN-CS-75-519, Memo AIM-266. Stanford, CA: Stanford University, Computer Science Department.

Eastman, C., 1973. "Automated Space Planning", *Artificial Intelligence*, Vol. 4, 1, pp. 41-64.

Erman, L. D. and Lesser, V. R., 1975. "A multi-level organization for problem-solving using many, diverse, cooperating sources of knowledge", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 483-490.

Fahlman, S. E., 1974. "A planning system for robot construction tasks", *Artificial Intelligence*, Vol. 5, 1, pp. 1-49.

Forgy, C. L., 1976. "A production system monitor for parallel computers", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science. In preparation.

Hayes-Roth, F., 1975. "Collected papers on the learning and recognition of structured patterns", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science. First paper in collection deals with representation.

Hedrick, C. L., 1974. "A computer program to learn production systems using a semantic net", Pittsburgh, PA: Carnegie-Mellon University, Graduate School of Industrial Administration. A shortened form is in *AI*, 7: 1, pp. 21-49, Spring, 1976.

McDermott, J., Newell, A. and Moore, J., 1976. "The efficiency of certain production system implementations", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Moore, J. and Newell, A., 1973. "How can MERLIN understand?", in Gregg, L., Ed., *Knowledge and Cognition*, pp. 201-252. Potomac, MD: Lawrence Erlbaum Associates.

Newell, A. and Simon, H. A., 1972. *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall.

Rumelhart, D. E., Lindsay, P. H. and Norman, D. A., 1972. "A process model for long-term memory", in Tulving, E. and Donaldson, W., Eds., *Organization and Memory*, New York, NY: Academic Press.

Rychener, M. D., 1975. "The Studnt production system: A study of encoding knowledge in production systems", Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Schank, R. C. and the Yale AI Project, 1975. "SAM -- a story understander", Research Report No. 43. New Haven, CT: Department of Computer Science, Yale University.

Simon, H. A., 1969. *The Sciences of the Artificial*, Cambridge, MA: The MIT Press.

Shapiro, S. C., 1971. "Net structure for semantic information storage, deduction, and retrieval", *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pp. 512-523. London.

Sussman, G. J., 1975. *A Computer Model of Skill Acquisition*, New York, NY: American Elsevier. Publication of Ph.D. thesis, MIT AI TR-297, 1973.

Winston, P. H., 1975. "Learning structural descriptions from examples", in Winston, P. H, Ed., *The Psychology of Computer Vision*, pp. 157-209. New York, NY: McGraw-Hill. Publication of Ph.D. thesis, MIT MAC TR-76, 1970.

CONCLUSION APPENDICES

Appendix A. RENAMINGS OF PREDICATES

ADD:LAST	ADD	•LINK/M
ADD:LINK	ADD	•LINK/A
APPLY:CHK	APPLY	•MOVE:OPR/C
APPLY:DIFFR	APPLY	•MOVE:OPR/E
APPLY:DIFFR:SETUP	APPLY	•MOVE:OPR/EC
APPLY:LOC:PROG	APPLY	•LOC:PROG/A
APPLY:OP	APPLY	•MOVE:OPR/A
APPLY:OP2	APPLY	•MOVE:OPR/H
APPLY:OPF	APPLY	•FORM:OPR/A
APPLY:RESULT	APPLY	•MOVE:OPR/R
ASSIGNS	VARBL	•ASG/T
ASSIGNS:D	VARBL	•ASG/M.2
ASSIGNS:N	VARBL	•ASG/M.1
CHANGES:VAL	MOVE:OPR	•VALCHANGE/T
CHECK:NUMV	GENRT	•DESASG/A.3
CHECK:RETRY	GOAL	•RETRY/W
CHECK:SAME	FILE	•GOAL/A.2
CHECK:SELX	GENRT	•DESASG/E
CHOOSE:OLD:GOAL	SELECT	•GOAL/S
CHOOSE:OLD:OBJ	SELECT	•OBJECT/S
COL:DANET	NETP	•DESASG/G
COL:LPNET	NETP	•LOC:PROG/G
COL:ONET	NETP	•OBJECT/G
COPY:LAST	COPY	•LINK/M
COPY:LINK	COPY	•LINK/A
COPY:OBJ	COPY	•OBJECT/A
DECR:LAST	DECR	•LINK/M
DECR:LINK	DECR	•LINK/A
DIFFR:EVAL	EVAL	•DIFFR/A
DIFFR:EVAL:RES1	EVAL	•DIFFR/A.1
DIFFR:EVAL:RES2	EVAL	•DIFFR/A.2
DIFFR:EVAL:RESULT	EVAL	•DIFFR/R
ERASE:APP	APPLY	•MOVE:OPR/DA
ERASE:CHOICES	SELECT	•GOAL/DS
ERASE:CHOICES:O	SELECT	•OBJECT/DS
ERASE:CS	FILE	•GOAL/DA.2
ERASE:CSP	FILE	•GOAL/DH.2
ERASE:LPC	LOC:PROG	•LINK/DT
ERASE:MATCH:DIFF	MATCH	/DA
ERASE:MD1	MATCH	•OBJECT/DA
ERASE:ML1	MATCH	/DA.2
ERASE:MN1	DIFFR	•LINK/DT
ERASE:MR1	MATCH	/DR
ERASE:MVAL	TRANSF	/DR.2
ERASE:OBJ	OBJECT	/DT
EVAL:GOAL	EVAL	•GOAL/A
EXT:DANET	FILE	•DESASG/H.1
EXT:DANET2	FILE	•DESASG/A.1
EXT:LPNET	FILE	•LOC:PROG/H.1
EXT:LPNET2	FILE	•LOC:PROG/A.1
EXT:ONET	FILE	•OBJECT/A.3
EXTREPR	OBJECT	•EXTREPR/G
FAIL	GOAL	/F
FAILED	GOAL	/DF
FEASASG	GENRT	•FEASASG/A
FILE:DES:ASG	FILE	•DESASG/A
FILE:GOAL	FILE	•GOAL/A
FILE:LOC:PROG	FILE	•LOC:PROG/A
FILE:OBJECT	FILE	•OBJECT/A
FORM2INPUT:METHOD	FORM2INPUT:OPR	/A
FORMOP:METHOD	FORM:OPR	/A
GENDES:ASG	GENRT	•DESASG/A.1
GENDES:ASG2	GENRT	•DESASG/A.2
GETLP:COMPON	LOC:PROG	•LINK/W
GPSRINIT	GPSR	/A
HASACTUAL:OBJ	GOAL	•OBJECT/T
HASALTDIFFR	GOAL	•ALTDIFFR/T
HASANTEC	GOAL	•ANTEC/T
HASDES:ASG	GOAL	•DESASG/T
HASDESIRED:OBJ	GOAL	•DESOBJECT/T
HASDIFFIC	GOAL	•DIFFIC/T
HASDIFFR	GOAL	•DIFFR/T
HASEXTREPR	OBJECT	•EXTREPR/T
HASLHS	NETP	•LHS/T
HASLINK	DIFFR	•LINK/T
HASLP:COMPON	LOC:PROG	•LINK/T
HASMOVE:COMPON	MOVE:OPR	•COMPON/T
HASNAME	DIFFR	•NAME/T
HASNEWFEAS	GOAL	•NEWFEAS/T
HASNEWFEAS:ORD	GOAL	•NEWFEAS/T.D
HASOP	GOAL	•OPR/T
HASOP:DIFFR	GOAL	•OPR:DIFFR
HASOP:DIFFR:ASG	APPLY	•MOVE:OPR/WM.3
HASREPR	NODE	•EXTREPR/T
HASSUPER:GOAL	GOAL	•SUPER/T
HASTOPNODE	OBJECT	•TOPNODE/T
HASTRACE:LEVEL	GOAL	•TRACE:LEVEL/T
HASVAL	NODE	•VALUE/T
HASVAR	COMPON	•VARBL/T
HASVARLINK	VARBL	•LINK/T
INCR:LAST	INCR	•LINK/M
INCR:LINK	INCR	•LINK/I
INSET	MOVE:OPR	•SET/T
IS2INPUT	FORM2INPUT:OPR	•TYPE/T
ISAPPLY:GOAL	GOAL	•TYPEAPPLY/T
ISDESCRIBED:OBJ	OBJECT	•TYPEDESCRIBED/T
ISDUMMY	OBJECT	•TYPEDUMMY/T
ISFORMOP	FORM:OPR	•TYPE/T
ISMOVEOP	MOVE:OPR	•TYPE/T
ISREDUCE:GOAL	GOAL	•TYPEREDUCE/T
ISSAME	OBJECT	•SAME/M
ISSAME:DA	VARBL	•ASG:SAME/M.2
ISSAME:EQV		OBJECT •SAME/T
ISSAME:GOAL	GOAL	•SAME/T
ISSET	SET	•TYPESET/T
ISTRANSFORM:GOAL	GOAL	•TYPETRANSFORM/T
LAST:DANET	NETP	•DESASG/M
LAST:LPNET	NETP	•LOC:PROG/M
LAST:ONET	NETP	•OBJECT/M
LINKS	NODE	•LINK/T
LOC:EXTR	MATCH	•OBJECT/A.2
LOC:PROG:RESULT	APPLY	•LOC:PROG/R
MATCH:DIF1	MATCH	•OBJECT/I
MATCH:DIFF	MATCH	•OBJECT/A
MATCH:RES1	MATCH	•OBJECT/R.1
MATCH:RESEXAM	FILE	•OBJECT/C.2
MATCH:RESTR	OBJECT	•RESTR/T
MATCH:RESULT	MATCH	•OBJECT/R
MATCH:VAL	TRANSF	/R.2
MATCH:VSET	TRANSF	/C.2
MCINIT	PROBLEM	•MC/A
METHODS:EXH	GOAL	•EXHAUST/T
MORE:DA	FILE	•DESASG/C
MOVEOP:METHOD	MOVE:OPR	•METHOD/A
NEXT:GOAL:APPLY	GOAL	•NEXT:APPLY/C
NEXT:GOAL:TRANS	GOAL	•NEXT:TRANS/C
ONET:SUCC	FILE	•OBJECT/A.5
ONET:SUCCH	FILE	•OBJECT/H.5
RECOG:GOAL	FILE	•GOAL/A.1
REDUCE:METHOD	REDUCE	/A
REDUCE:OPCHK	REDUCE	•FORM/C
REM:LAST	REMOVE	•LINK/M
REM:LINK	REMOVE	•LINK/A
RESULT:SETUP	MATCH	•OBJECT/C.3
RETRY	GOAL	•RETRY/T
RETRY:TRANS	TRANSF	•RETRY/T
SELECT:DESASG	GENRT	•DESASG/A
SELECT:METHOD	SELECT	•METHOD/A
SELECT:NEW:OBJ	SELECT	•OBJECT/T
SELECT:OP	REDUCE	/A.1
SPLIT:OB	FILE	•OBJECT/A.4
SPROUT:RED:APP	GOAL	•REDUCE•NEXT:APPLY/W
SPROUT:RED:TRANS	GOAL	•REDUCE•NEXT:TRANS/W
SUCCEED	GOAL	/T
SUCCEEDED	GOAL	/DT
TEST:ONET	NETP	•OBJECT/A
TEST:ONET:T	FILE	•OBJECT/H.1
TEST:ONET:R	FILE	•OBJECT/A.1
TEST:ONET:S	FILE	•OBJECT/A.2
TRACE:ASG	TRACE	•DESASG/A
TRACE:GOAL	TRACE	•GOAL/A
TRACE:IND	TRACE	•INDENT/T
TRACE:OBJ	TRACE	•OBJECT/A
TRACING	TRACE	/T
TRANSF2	TRANSF	/H.3
TRANSF3	TRANSF	/A.3
TRANSFORM:METHOD	TRANSF	/A
TRY:OLD:GOALS	SELECT	•OLDGOALOBJECT/A
TRYAPP	APPLY	•MOVE:OPR/WA
TRYAPP2	APPLY	•MOVE:OPR/WA.A
TRYAPP:DIFFR:SETUP	APPLY	•MOVE:OPR/WC.3
TRYAPP:RESULT	APPLY	•MOVE:OPR/WE
TRYAPPH	APPLY	•MOVE:OPR/WH

A.

VARDOMAIN	VARBL	•DOMAIN/T
XRCOLL	NODE	•EXTREPR/G

- - - - - - - - - - - - - - - - - - - -

ADD		
	•LINK/A	ADDLINK
	•LINK/M	ADDLAST
APPLY		
	•MOVE:OPR/WH	TRYAPPH
	•MOVE:OPR/WE	TRYAPP:RESULT
	•MOVE:OPR/WC.3	TRYAPP:DIFFR:SETUP
	•MOVE:OPR/WA.4	TRYAPP2
	•MOVE:OPR/WA	TRYAPP
	•LOC:PROG/R	LOC:PROG:RESULT
	•MOVE:OPR/WM.3	HASOP:DIFFR:ASG
	•MOVE:OPR/DA	ERASE:APP
	•MOVE:OPR/R	APPLY:RESULT
	•FORM:OPR/A	APPLY:OPF
	•MOVE:OPR/M	APPLY:OP2
	•MOVE:OPR/A	APPLY:OP
	•LOC:PROG/A	APPLY:LOC:PROG
	•MOVE:OPR/EC	APPLY:DIFFR:SETUP
	•MOVE:OPR/E	APPLY:DIFFR
	•MOVE:OPR/C	APPLY:CHK
COMPON		
	•VARBL/T	HASVAR
COPY		
	•OBJECT/A	COPY:OBJ
	•LINK/A	COPYLINK
	•LINK/M	COPYLAST
DECR		
	•LINK/A	DECRLINK
	•LINK/M	DECRLAST
DIFFR		
	•NAME/T	HASNAME
	•LINK/T	HASLINK
	•LINK/DT	ERASE:MN1
EVAL		
	•GOAL/A	EVAL:GOAL
	•DIFFR/R	DIFFR:EVAL:RESULT
	•DIFFR/A.2	DIFFR:EVAL:RES2
	•DIFFR/A.1	DIFFR:EVAL:RES1
	•DIFFR/A	DIFFR:EVAL
FILE		
	•OBJECT/A.2	TEST:ONE:TS
	•OBJECT/A.1	TEST:ONE:TR
	•OBJECT/M.1	TEST:ONE:TF
	•OBJECT/A.4	SPLIT:OB
	•GOAL/A.1	RECOG:GOAL
	•OBJECT/M.5	ONET:SUCCH
	•OBJECT/A.5	ONET:SUCC
	•DESASG/C	MORE:DA
	•OBJECT/C.2	MATCH:RES:EXAM
	•OBJECT/A	FILE:OBJECT
	•LOC:PROG/A	FILE:LOC:PROG
	•GOAL/A	FILE:GOAL
	•DESASG/A	FILE:DES:ASG
	•OBJECT/A.3	EXT:ONET
	•LOC:PROG/A.1	EXT:LPNET2
	•LOC:PROG/M.1	EXT:LPNET
	•DESASG/A.1	EXT:DANET2
	•DESASG/M.1	EXT:DANET
	•GOAL/DM.2	ERASE:CSP
	•GOAL/DA.2	ERASE:CS
	•GOAL/A.2	CHECK:SAME
FORM2INPUT:OPR		
	•TYPE/T	IS2INPUT
	/A	FORM2INPUT:METHOD
FORM:OPR		
	•TYPE/T	ISFORMOP
	/A	FORMOP:METHOD
GENRT		
	•DESASG/A	SELECT:DES:ASG
	•DESASG/A.2	GEN:DES:ASG2
	•DESASG/A.1	GEN:DES:ASG
	•FEASASG/A	FEASASG
	•DESASG/E	CHECK:SELX
	•DESASG/A.3	CHECK:NUMV
GOAL		
	/DT	SUCCEEDED
	/T	SUCCEED
	•REDUCE•NEXT:TRANS/W	SPROUT:RED:TRANS
	•REDUCE•NEXT:APPLY/W	SPROUT:RED:APP
	•RETRY/T	RETRY
	•NEXT:TRANS/C	NEXT:GOAL:TRANS
	•NEXT:APPLY/C	NEXT:GOAL:APPLY
	•EXHAUST/T	METHODS:EXH
	•TYPETRANSFORM/T	ISTRANSFORM:GOAL
	•SAME/T	ISSAME:GOAL
	•TYPEREDUCE/T	ISREDUCE:GOAL
	•TYPEAPPLY/T	ISAPPLY:GOAL
	•TRACE:LEVEL/T	HASTRACE:LEVEL
	•SUPER/T	HASSUPER:GOAL
	•OPRDIFFR	HASOP:DIFFR
	•OPR/T	HASOP
	•NEWFEAS/T.0	HASNEWFEAS:ORD
	•NEWFEAS/T	HASNEWFEAS
	•DIFFR/T	HASDIFFR
	•DIFFIC/T	HASDIFFIC
	•DESOBJECT/T	HASDESIRED:OBJ
	•DESASG/T	HASDES:ASG
	•ANTEC/T	HASANTEC
	•ALTDIFFR/T	HASALT:DIFFR
	•OBJECT/T	HASACTUAL:OBJ
	/DF	FAILED
	/F	FAIL
	•RETRY/W	CHECK:RETRY
GPSR		
	/A	GPSRINIT
INCR		
	•LINK/I	INCRLINK
	•LINK/M	INCRLAST
LOC:PROG		
	•LINK/T	HASLP:COMPON
	•LINK/W	GETLP:COMPON
	•LINK/DT	ERASE:LPC
MATCH		
	•OBJECT/C.3	RESULT:SETUP
	•OBJECT/R	MATCH:RESULT
	•OBJECT/R.1	MATCH:RES1
	•OBJECT/A	MATCH:DIFF
	•OBJECT/I	MATCH:DIF1
	•OBJECT/A.2	LOC:EXTR
	/DR	ERASE:MR1
	/DA.2	ERASE:ML1
	•OBJECT/DA	ERASE:MD1
	/DA	ERASE:MATCH:DIFF
MOVE:OPR		
	•METHOD/A	MOVEOP:METHOD
	•TYPE/T	ISMOVEOP
	•SET/T	INSET
	•COMPON/T	HASMOVE:COMPON
	•VALCHANGE/T	CHANGES:VAL
NETP		
	•OBJECT/A	TEST:ONET
	•OBJECT/M	LAST:ONET
	•LOC:PROG/M	LAST:LPNET
	•DESASG/M	LAST:DANET
	•LHS/T	HASLHS
	•OBJECT/G	COL:ONET
	•LOC:PROG/G	COL:LPNET
	•DESASG/G	COL:DANET
NODE		
	•EXTREPR/G	XRCOLL
	•LINK/T	LINKS
	•VALUE/T	HASVAL
	•EXTREPR/T	HASREPR
OBJECT		
	•RESTR/T	MATCH:RESTR
	•SAME/T	ISSAME:EQV
	•SAME/M	ISSAME
	•TYPEDUMMY/T	ISDUMMY
	•TYPEDESCRIBED/T	ISDESCRIBED:OBJ
	•TOPNODE/T	HASTOPNODE
	•EXTREPR/T	HASEXTREPR
	•EXTREPR/G	EXTREPR
	/DT	ERASE:OBJ
PROBLEM		
	•MC/A	MCINIT
PROD		
	•LHS/T	HASLHS
REDUCE		
	/A.1	SELECTOP

A.

	•FORM/C	REDUCE:OPCHK
	/A	REDUCEMETHOD
REMOVE		
	•LINK/A	REMLINK
	•LINK/M	REMLAST
SELECT		
	•OLDGOALOBJECT/A	TRY:OLDGOALS
	•OBJECT/T	SELECTNEW:OBJ
	•METHOD/A	SELECTMETHOD
	•OBJECT/DS	ERASE:CHOICES:O
	•GOAL/DS	ERASE:CHOICES
	•OBJECT/S	CHOOSE:OLD:OBJ
	•GOAL/S	CHOOSE:OLD:GOAL
SET		
	•TYPESET/T	ISSET
TRACE		
	/T	TRACING
	•OBJECT/A	TRACE:OBJ
	•INDENT/T	TRACE:IND
	•GOAL/A	TRACE:GOAL
	•DESASG/A	TRACE:ASG
TRANSF		
	/A	TRANSFORMMETHOD
	/A.3	TRANSF3
	/M.3	TRANSF2
	•RETRY/T	RETRY:TRANS
	/C.2	MATCH:VSET
	/R.2	MATCH:VAL
	/DR.2	ERASE:MVAL
VARBL		
	•ASG:SAME/M.2	ISSAME:DA
	•ASG/M.1	ASSIGNS:N
	•ASG/M.2	ASSIGNS:O
	•ASG/T	ASSIGNS
	•DOMAIN/T	VARDOMAIN
	•LINK/T	HASVARLINK

Appendix B. FIRST ABSTRACTION

```
E0 " PROB INIT " (GPSR /A)
  •>
 (OBJECT •TYPEDUMMY/T) (OBJECT •TOPNODE/T) (TRACE •INDENT/T)
 (NETP •LOCPROG/M) (NETP •OBJECT/M) (NETP •DESASG/M) (GOAL •TRACELEVEL/T)
 (NOT (GPSR /A))

E1 " GOAL EVAL OK " (EVAL •GOAL/A) (NOT ((GOAL •SAME/T)))
 (NOT ((GOAL •DIFFIC/T)))
  •>
 (SELECT •METHOD/A) (NOT (EVAL •GOAL/A))

E2 " GOAL EVAL- " (EVAL •GOAL/A) (NOT (FILE •GOAL/A.1)) (GOAL •DIFFIC/T)
 (GOAL •TYPEREDUCE/T) (NOT ((GOAL •SAME/T))) (GOAL •TRACELEVEL/T)
 (GOAL •SUPER/T)
  •>
 (TRACE /T) (GOAL •RETRY/W) (NOT (EVAL •GOAL/A))

E2R " GOAL EVAL-R " (EVAL •GOAL/A) (NOT (FILE •GOAL/A.1)) (GOAL •DIFFIC/T)
 (GOAL •TYPEREDUCE/T) (GOAL •SAME/T) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (SELECT •OLDGOALOBJECT/A) (NOT (EVAL •GOAL/A))
 (NOT (GOAL •DIFFIC/T)) (NOT (GOAL •TRACELEVEL/T))

E3 " GOAL EVAL-A " (EVAL •GOAL/A) (NOT (FILE •GOAL/A.1)) (GOAL •DIFFIC/T)
 (GOAL •ANTEC/T) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (GOAL /F) (GOAL •EXHAUST/T) (NOT (EVAL •GOAL/A))

E4 " GOAL EVAL-S " (EVAL •GOAL/A) (NOT (FILE •GOAL/A.1)) (GOAL •DIFFIC/T)
 (NOT ((GOAL •ANTEC/T))) (GOAL •TRACELEVEL/T) (GOAL •SUPER/T)
 (GOAL •TYPEREDUCE/T)
  •>
 (TRACE /T) (GOAL /F) (GOAL •EXHAUST/T) (NOT (EVAL •GOAL/A))

E8 " SAME REP " (EVAL •GOAL/A) (GOAL •SAME/T) (GOAL •TRACELEVEL/T)
 (GOAL •DIFFIC/T)
  •>
 (TRACE /T) (SELECT •OLDGOALOBJECT/A) (NOT (EVAL •GOAL/A))
 (NOT (GOAL •TRACELEVEL/T)) (NOT (GOAL •DIFFIC/T))

E10 " SUC TRANS " (GOAL /T) (GOAL •NEXT:TRANS/C) (GOAL •DIFFIC/T)
 (GOAL •SUPER/T) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (FILE •GOAL/A) (EVAL •GOAL/A) (GOAL •TYPETRANSFORM/T)
 (GOAL •OBJECT/T) (GOAL •DESOBJECT/T) (GOAL •ANTEC/T) (GOAL •SUPER/T)
 (GOAL •DIFFIC/T) (GOAL /OT) (NOT (GOAL /T))

E11 " SUC APPLY " (GOAL /T) (GOAL •NEXT:APPLY/C) (GOAL •SUPER/T)
 (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (FILE •GOAL/A) (EVAL •GOAL/A) (GOAL •TYPEAPPLY/T) (GOAL •OBJECT/T)
 (GOAL •DIFFIC/T) (GOAL •DESASG/T) (GOAL •SUPER/T) (GOAL •OPR/T)
 (GOAL •ANTEC/T) (GOAL /OT) (NOT (GOAL /T))

E12 " SUC SUP " (GOAL /T) (NOT ((GOAL •NEXT:TRANS/C)))
 (NOT ((GOAL •NEXT:APPLY/C))) (GOAL •SUPER/T) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (GOAL /T) (GOAL /OT)

E20 " FAIL ANTEC " (GOAL /F) (GOAL •ANTEC/T) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (GOAL •RETRY/W) (GOAL /OF) (NOT (GOAL /F))

E21 " FAIL ANTEC- " (GOAL /F) (NOT ((GOAL •ANTEC/T))) (GOAL •SUPER/T)
 (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (GOAL •RETRY/W) (GOAL /OF) (NOT (GOAL /F))

E22 " CHECK RETRY- " (GOAL •RETRY/W) (NOT (GOAL •EXHAUST/T))
 (NOT (GOAL •TYPETRANSFORM/T)) (GOAL •TRACELEVEL/T)
  •>
 (TRACE /T) (SELECT •METHOD/A) (GOAL •RETRY/T) (NOT (GOAL •RETRY/W))

E23 " CHECK RETRY EXH " (GOAL •RETRY/W) (GOAL •EXHAUST/T) (NOT (GOAL /OF))
  •>
 (GOAL /F) (NOT (GOAL •RETRY/W))

E23R " FAIL REP " (GOAL •RETRY/W) (GOAL /OF) (TRACE •INDENT/T)
  •>
 (TRACE /T) (SELECT •OLDGOALOBJECT/A) (NOT (GOAL •RETRY/W))
```

B.

```
E24 " CHECK RETRY TG " (GOAL •RETRY/W) (GOAL •TYPETRANSFORM/T)
 (NOT ((TRANSF •RETRY/T))) (GOAL •TRACE&LEVEL/T)
  •>
 (TRACE /T) (SELECT •OLDGOALOBJECT/A) (NOT (GOAL •RETRY/W))

E25 " RETRY TRANS " (GOAL •RETRY/W) (GOAL •TYPETRANSFORM/T) (TRANSF •RETRY/T)
 (GOAL •ALTDIFFR/T) (GOAL •TRACE&LEVEL/T)
  •>
 (TRANSF /A.3) (TRANSF /R.2) (TRACE /T) (NOT (GOAL •RETRY/W))
 (NOT (GOAL •ALTDIFFR/T))

E26 " RETRY TRANS- " (GOAL •RETRY/W) (GOAL •TYPETRANSFORM/T) (TRANSF •RETRY/T)
 (NOT ((GOAL •ALTDIFFR/T))) (GOAL •TRACE&LEVEL/T)
  •>
 (TRACE /T) (SELECT •OLDGOALOBJECT/A) (NOT (GOAL •RETRY/W))

E30 " TRY OLD GOALS " (SELECT •OLDGOALOBJECT/A) (GOAL •TYPEREDUCE/T)
 (NOT
  ((SELECT •OBJECT/T) (GOAL •OBJECT/T) (NOT (GOAL •TYPETRANSFORM/T))
     (NOT ((GOAL •OBJECT/T) (GOAL •TYPETRANSFORM/T)))))
 (NOT (GOAL •EXHAUST/T)) (GOAL •DIFFIC/T)
 (NOT ((GOAL •TYPEREDUCE/T) (NOT (GOAL •EXHAUST/T)) (GOAL •DIFFIC/T)))
 (GOAL •SUPER/T)
 (NOT
  ((GOAL •TYPEAPPLY/T) (GOAL •TYPEREDUCE/T) (GOAL •DIFFIC/T)
     (NOT (GOAL •EXHAUST/T)) (GOAL •SUPER/T) (GOAL •TYPETRANSFORM/T)))
  •>
 (SELECT •GOAL/S) (NOT (SELECT •OLDGOALOBJECT/A))

E31 " CHOOSE OLD " (SELECT •GOAL/S) (NOT ((SELECT •GOAL/S)))
 (GOAL •TRACE&LEVEL/T) (TRACE •INDENT/T)
  •>
 (SELECT •GOAL/DS) (TRACE /T) (SELECT •METHOD/A) (GOAL •RETRY/T)
 (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

E32 " ERASE CH " (SELECT •GOAL/DS) (SELECT •GOAL/S)
  •>
 (NOT (SELECT •GOAL/DS)) (NOT (SELECT •GOAL/S))

E35 " NEW OBJ CRIT " (SELECT •OLDGOALOBJECT/A) (SELECT •OBJECT/T)
 (GOAL •OBJECT/T) (NOT (GOAL •TYPETRANSFORM/T))
 (NOT ((GOAL •OBJECT/T) (GOAL •TYPETRANSFORM/T)))
  •>
 (SELECT •OBJECT/S) (NOT (SELECT •OLDGOALOBJECT/A))

E36 " CHOOSE OBJ " (SELECT •OBJECT/S) (NOT ((SELECT •OBJECT/S)))
 (TRACE •INDENT/T) (GOAL •SUPER/T) (GOAL •DESOBJECT/T) (NOT ((GOAL •SUPER/T)))
  •>
 (SELECT •OBJECT/DS) (TRACE /T) (FILE •GOAL/A) (EVAL •GOAL/A)
 (GOAL •TYPETRANSFORM/T) (GOAL •OBJECT/T) (GOAL •DESOBJECT/T) (GOAL •SUPER/T)
 (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

E37 " ERASE CH " (SELECT •OBJECT/DS) (SELECT •OBJECT/S)
  •>
 (NOT (SELECT •OBJECT/DS)) (NOT (SELECT •OBJECT/S))

F1 " FILE LOC-PROG " (FILE •LOC-PROG/A) (DIFFR •LINK/T) (NOT ((DIFFR •LINK/T)))
  •>
 (DIFFR •LINK/T) (FILE •LOC-PROG/H.1) (NOT (FILE •LOC-PROG/A))

F2 " EXTEND LP NET " (FILE •LOC-PROG/H.1)
  •>
 (FILE •LOC-PROG/A.1) (NOT (FILE •LOC-PROG/H.1))

F3 " ST LP NET COL " (FILE •LOC-PROG/A.1) (DIFFR •LINK/T)
 (NOT ((DIFFR •LINK/T)))
  •>
 (METP •LOC-PROG/G) (LOC-PROG •LINK/T) (NOT (FILE •LOC-PROG/A.1))
 (NOT (DIFFR •LINK/T))

F4 " COL LP NET " (METP •LOC-PROG/G) (DIFFR •LINK/T) (NOT ((DIFFR •LINK/T)))
  •>
 (METP •LOC-PROG/G) (LOC-PROG •LINK/T) (NOT (DIFFR •LINK/T))

F5 " COL LP NET D " (METP •LOC-PROG/G) (NOT ((DIFFR •LINK/T)))
 (METP •LOC-PROG/M) (TRACE •INDENT/T)
  •>
 (METP •LOC-PROG/M) (DIFFR •NAME/T) (CHANGE-PM Z) (TRACE /T)
 (OBJECT •EXTREPR/T) (NOT (METP •LOC-PROG/G))

F5E " LAST MLT " (METP •LOC-PROG/M)
  •>
 (METP •LOC-PROG/M)
```

```
F6 " FILE GOAL " (FILE •GOAL/A)
  •>
 (TRACE •GOAL/A) (FILE •GOAL/A.1) (NOT (FILE •GOAL/A))

F7 " REC GT- " (FILE •GOAL/A.1) (GOAL •TYPETRANSFORM/T) (GOAL •DESOBJECT/T)
 (GOAL •OBJECT/T)
 (NOT ((GOAL •OBJECT/T) (GOAL •TYPETRANSFORM/T) (GOAL •DESOBJECT/T)))
  •>
 (NOT (FILE •GOAL/A.1))

F7N " REC GT " (FILE •GOAL/A.1) (GOAL •TYPETRANSFORM/T) (GOAL •DESOBJECT/T)
 (GOAL •OBJECT/T)
 (NOT ((GOAL •OBJECT/T) (GOAL •TYPETRANSFORM/T) (GOAL •DESOBJECT/T)))
  •>
 (GOAL •SAME/T) (NOT (FILE •GOAL/A.1))

F8 " REC GA- " (FILE •GOAL/A.1) (GOAL •TYPEAPPLY/T) (GOAL •OBJECT/T)
 (GOAL •OPR/T) (GOAL •DESASG/T)
 (NOT ((GOAL •OBJECT/T) (GOAL •DESASG/T) (GOAL •TYPEAPPLY/T) (GOAL •OPR/T)))
  •>
 (NOT (FILE •GOAL/A.1))

F8N " REC GA " (FILE •GOAL/A.1) (GOAL •TYPEAPPLY/T) (GOAL •OBJECT/T)
 (GOAL •OPR/T) (GOAL •DESASG/T)
  •>
 (FILE •GOAL/A.2) (FILE •GOAL/DH.2) (NOT (FILE •GOAL/A.1))

F8S " OLD NO DIFFR " (FILE •GOAL/A.2) (NOT ((GOAL •OPRDIFFR)))
 (NOT ((FILE •GOAL/A.2) (NOT ((GOAL •OPRDIFFR)))))
  •>
 (FILE •GOAL/DA.2) (GOAL •SAME/T) (NOT (FILE •GOAL/DH.2))

F8T " OLD DIFFR + " (FILE •GOAL/A.2) (GOAL •OPRDIFFR)
 (NOT ((FILE •GOAL/A.2) (GOAL •OPRDIFFR)))
  •>
 (FILE •GOAL/DA.2) (GOAL •SAME/T) (NOT (FILE •GOAL/DH.2))

F8Y " ERS CSP " (FILE •GOAL/DH.2)
  •>
 (FILE •GOAL/DA.2) (NOT (FILE •GOAL/DH.2))

F8Z " ERS CS " (FILE •GOAL/DA.2) (FILE •GOAL/A.2)
  •>
 (NOT (FILE •GOAL/DA.2)) (NOT (FILE •GOAL/A.2))

F9 " REC GR- " (FILE •GOAL/A.1) (GOAL •TYPEREDUCE/T) (GOAL •OBJECT/T)
 (GOAL •DIFFR/T) (NOT ((GOAL •DIFFR/T) (GOAL •OBJECT/T) (GOAL •TYPEREDUCE/T)))
  •>
 (NOT (FILE •GOAL/A.1))

F9N " REC GR " (FILE •GOAL/A.1) (GOAL •TYPEREDUCE/T) (GOAL •DIFFR/T)
 (GOAL •OBJECT/T) (NOT ((GOAL •DIFFR/T) (GOAL •OBJECT/T) (GOAL •TYPEREDUCE/T)))
  •>
 (GOAL •SAME/T) (NOT (FILE •GOAL/A.1))

F10 " FILE OBJECT " (FILE •OBJECT/A)
  •>
 (OBJECT •EXTREPR/G) (METP •OBJECT/A) (FILE •OBJECT/H.1)
 (NOT (FILE •OBJECT/A))

F11 " TEST FIN " (FILE •OBJECT/H.1)
  •>
 (FILE •OBJECT/A.1) (NOT (FILE •OBJECT/H.1)) (NOT (METP •OBJECT/A))

F13 " NEW MET PROD " (FILE •OBJECT/A.1) (NOT ((OBJECT •SAME/M)))
 (OBJECT •TYPEDUMMY/T) (METP •OBJECT/M)
  •>
 (MATCH •OBJECT/A) (FILE •OBJECT/C.2) (MATCH •OBJECT/I) (CHANGE-PM Z)
 (METP •LHS/T) (METP •OBJECT/M) (NOT (FILE •OBJECT/A.1))

F14 " SAME SET " (FILE •OBJECT/A.1) (OBJECT •SAME/M)
  •>
 (FILE •OBJECT/A.2) (NOT (FILE •OBJECT/A.1)) (NOT (OBJECT •SAME/M))

F15 " SAME OBJ " (FILE •OBJECT/A.2) (NOT ((OBJECT •SAME/T)))
  •>
 (MATCH •OBJECT/A) (FILE •OBJECT/C.2) (MATCH •OBJECT/I)
 (NOT (FILE •OBJECT/A.2))

F17 " SAME EQV " (FILE •OBJECT/A.2) (OBJECT •SAME/T)
  •>
 (MATCH •OBJECT/A) (FILE •OBJECT/C.2) (MATCH •OBJECT/I)
 (NOT (FILE •OBJECT/A.2))
```

```
F19 " LAST MLT " (NETP =OBJECT/M)
   =>
 (NETP =OBJECT/M)

F20 " OBJ DIFR " (FILE =OBJECT/C.2) (MATCH =OBJECT/R.1)
 (NOT ((MATCH =OBJECT/R.1)))
   =>
 (MATCH =OBJECT/DA) (MATCH /DR) (MATCH /DA.2) (FILE =OBJECT/A.3)
 (NOT (FILE =OBJECT/C.2)) (NOT (MATCH =OBJECT/R.1))

F21 " ERS MDI " (MATCH =OBJECT/DA) (MATCH =OBJECT/A)
   =>
 (MATCH =OBJECT/I) (NOT (MATCH =OBJECT/DA)) (NOT (MATCH =OBJECT/A))

F22 " ERS MRI " (MATCH /DR) (MATCH =OBJECT/R.1)
   =>
 (DIFFR =LINK/DT) (NOT (MATCH /DR)) (NOT (MATCH =OBJECT/R.1))

F23 " ERS MRI- " (MATCH /DR) (NOT ((MATCH =OBJECT/R.1)))
   =>
 (NOT (MATCH /DR))

F24 " ERS MLI " (MATCH /DA.2) (MATCH =OBJECT/A.2)
   =>
 (DIFFR =LINK/DT) (NOT (MATCH /DA.2)) (NOT (MATCH =OBJECT/A.2))

F25 " ERS MLI- " (MATCH /DA.2) (NOT ((MATCH =OBJECT/A.2)))
   =>
 (NOT (MATCH /DA.2))

F26 " ERS MNI " (DIFFR =LINK/DT) (DIFFR =LINK/T)
   =>
 (NOT (DIFFR =LINK/DT)) (NOT (DIFFR =LINK/T))

F27 " ERS MNI- " (DIFFR =LINK/DT) (NOT ((DIFFR =LINK/T)))
   =>
 (NOT (DIFFR =LINK/DT))

F28 " ERS MDI SIG " (MATCH =OBJECT/I) (NOT ((MATCH =OBJECT/A)))
   =>
 (NOT (MATCH =OBJECT/I))

F30 " EXT ON ST " (FILE =OBJECT/A.3) (NETP =LHS/T) (DIFFR =LINK/T)
 (NOT ((DIFFR =LINK/T)))
   =>
 (NETP =OBJECT/G) (NOT (FILE =OBJECT/A.3)) (NOT (DIFFR =LINK/T))

F32 " EXT ON " (NETP =OBJECT/G) (DIFFR =LINK/T) (NOT ((DIFFR =LINK/T)))
   =>
 (NETP =OBJECT/G) (NOT (DIFFR =LINK/T))

F34 " SPLIT ON P " (NETP =OBJECT/G) (NOT (OBJECT =TYPEDUMMY/T))
 (NOT ((DIFFR =LINK/T))) (NETP =LHS/T) (NETP =OBJECT/M)
 (NOT ((NETP =OBJECT/M)))
   =>
 (FILE =OBJECT/A.4) (CHANGEPM 2) (NETP =OBJECT/M) (NOT (NETP =OBJECT/G))
 (NOT (NETP =LHS/T))

F35 " SPLIT ON P DM " (NETP =OBJECT/G) (OBJECT =TYPEDUMMY/T)
 (NOT ((DIFFR =LINK/T))) (NETP =LHS/T)
   =>
 (FILE =OBJECT/A.4) (CHANGEPM 2) (NOT (NETP =OBJECT/G)) (NOT (NETP =LHS/T))

F36 " SPLIT OB1 " (FILE =OBJECT/A.4)
   =>
 (CHANGEPM 2) (NOT (FILE =OBJECT/A.4))

F38 " SPLIT OB2 " (FILE =OBJECT/A.4)
   =>
 (CHANGEPM 2) (NOT (FILE =OBJECT/A.4))

F40 " NO DIFFR " (FILE =OBJECT/C.2) (NOT ((MATCH =OBJECT/R.1))) (NETP =LHS/T)
 (TRACE =INDENT/T)
   =>
 (MATCH =OBJECT/DA) (OBJECT =SAME/T) (TRACE /T) (NOT (FILE =OBJECT/C.2))
 (NOT (NETP =LHS/T))

F41 " CHECK DONE " (OBJECT =SAME/T) (GOAL /T)
   =>
 (FILE =OBJECT/A.5) (NOT (GOAL /T))

F42 " ONET DONE " (FILE =OBJECT/A.5) (NOT ((MATCH =OBJECT/A)))
 (NOT ((MATCH =OBJECT/R.1))) (NOT ((MATCH =OBJECT/A.2)))
 (NOT ((FILE =OBJECT/A.2)))
   =>
 (OBJECT /DT) (GOAL /T) (NOT (FILE =OBJECT/A.5))

F42D " ONET CONT D " (FILE =OBJECT/A.5) (MATCH =OBJECT/A)
 (NOT ((MATCH =OBJECT/R.1)))
   =>
 (MATCH =OBJECT/A) (FILE =OBJECT/H.5) (NOT (FILE =OBJECT/A.5))

F42L " ONET CONT L " (FILE =OBJECT/A.5) (MATCH =OBJECT/A.2)
 (NOT ((MATCH =OBJECT/R.1)))
   =>
 (MATCH =OBJECT/A.2) (FILE =OBJECT/H.5) (NOT (FILE =OBJECT/A.5))

F42R " ONET CONT R " (FILE =OBJECT/A.5) (MATCH =OBJECT/R.1)
   =>
 (MATCH =OBJECT/R.1) (FILE =OBJECT/H.5) (NOT (FILE =OBJECT/A.5))

F42S " ONET CONT S " (FILE =OBJECT/A.5) (FILE =OBJECT/A.2)
   =>
 (FILE =OBJECT/A.2) (FILE =OBJECT/H.5) (NOT (FILE =OBJECT/A.5))

F42U " UN-H OS " (FILE =OBJECT/H.5)
   =>
 (FILE =OBJECT/A.5) (NOT (FILE =OBJECT/H.5))

F44 " ERS OBJ " (OBJECT /DT) (OBJECT =TOPNODE/T) (OBJECT =EXTREPR/T)
   =>
 (OBJECT /DT) (NOT (OBJECT =TOPNODE/T)) (NOT (OBJECT =EXTREPR/T))

F46 " ERS OBJ N " (OBJECT /DT) (NODE =LINK/T) (NOT ((NODE =VALUE/T)))
   =>
 (OBJECT /DT) (NOT (NODE =LINK/T))

F47 " ERS OBJ NV " (OBJECT /DT) (NODE =LINK/T) (NODE =VALUE/T)
   =>
 (NOT (OBJECT /DT)) (NOT (NODE =LINK/T)) (NOT (NODE =VALUE/T))

F48 " ERS OBJ N- " (OBJECT /DT) (NOT ((NODE =LINK/T)))
 (NOT ((OBJECT =TOPNODE/T)))
   =>
 (NOT (OBJECT /DT))

F50 " FILE DES ASG " (FILE =DESASG/A) (VARBL =ASG/M.1) (NOT ((VARBL =ASG/M.1)))
   =>
 (VARBL =ASG/M.1) (FILE =DESASG/H.1) (NOT (FILE =DESASG/A))

F51 " OLD DA " (VARBL =ASGSAME/M.2) (FILE =DESASG/H.1) (GOAL =DESASG/T)
 (GENRT =FEASASG/A)
   =>
 (FILE =DESASG/C) (GENRT =FEASASG/A) (GOAL =DESASG/T) (NOT (VARBL =ASGSAME/M.2))
 (NOT (FILE =DESASG/H.1))

F51M " MORE DA " (FILE =DESASG/C) (FILE =DESASG/H.1)
   =>
 (FILE =DESASG/A) (NOT (FILE =DESASG/C))

F51N " MORE DA- " (FILE =DESASG/C) (NOT ((FILE =DESASG/H.1)))
   =>
 (NOT (FILE =DESASG/C))

F52 " EXTEND DA MET " (FILE =DESASG/H.1)
   =>
 (FILE =DESASG/A.1) (NOT (FILE =DESASG/H.1))

F53 " DA MET COL " (FILE =DESASG/A.1) (VARBL =ASG/M.1) (NOT ((VARBL =ASG/M.1)))
 (NOT ((VARBL =ASG/M.1)))
   =>
 (NETP =DESASG/G) (VARBL =ASG/M.2) (NOT (FILE =DESASG/A.1)) (NOT (VARBL =ASG/M.1))

F55 " COL DA MET " (NETP =DESASG/G) (VARBL =ASG/M.1) (NOT ((VARBL =ASG/M.1)))
 (NOT ((VARBL =ASG/M.1)))
   =>
 (NETP =DESASG/G) (VARBL =ASG/M.2) (NOT (VARBL =ASG/M.1))

F56 " COL DA MET D " (NETP =DESASG/G) (NOT ((VARBL =ASG/M.1))) (NETP =DESASG/M)
   =>
 (NETP =DESASG/M) (CHANGEPM 2) (NOT (NETP =DESASG/G))

F57 " LAST MLT " (NETP =DESASG/M)
   =>
 (NETP =DESASG/M)
```

```
M1 " SEL TF " (SELECT •METHOD/A) (GOAL •TYPETRANSFORM/T)
  •>
 (TRANSF /A) (NOT (SELECT •METHOD/A))

M2 " SEL RED " (SELECT •METHOD/A) (GOAL •TYPEREDUCE/T)
  •>
 (REDUCE /A) (NOT (SELECT •METHOD/A))

M3 " SEL APPM " (SELECT •METHOD/A) (GOAL •TYPEAPPLY/T) (GOAL •OPR/T)
 (MOVE:OPR •TYPE/T)
  •>
 (MOVE:OPR •METHOD/A) (NOT (SELECT •METHOD/A))

M4 " SEL APPF1 " (SELECT •METHOD/A) (GOAL •TYPEAPPLY/T) (GOAL •OPR/T)
 (FORM:OPR •TYPE/T) (NOT (FORM2INPUT:OPR •TYPE/T))
  •>
 (FORM:OPR /A) (NOT (SELECT •METHOD/A))

M5 " SEL APPF2 " (SELECT •METHOD/A) (GOAL •TYPEAPPLY/T) (GOAL •OPR/T)
 (FORM:OPR •TYPE/T) (FORM2INPUT:OPR •TYPE/T)
  •>
 (FORM2INPUT:OPR /A) (NOT (SELECT •METHOD/A))

M20 " TRANSF:G " (TRANSF /A) (GOAL •OBJECT/T) (GOAL •DESOBJECT/T)
  •>
 (MATCH •OBJECT/A) (TRANSF /H.3) (NOT (TRANSF /A))

M20S " SUC TRANS " (TRANSF /A) (GOAL •OBJECT/T) (GOAL •DESOBJECT/T)
  •>
 (GOAL /T) (NOT (TRANSF /A))

M21 " MATCH RESULT " (MATCH •OBJECT/R)
  •>
 (EVAL •DIFFR/A) (TRANSF /C.2) (NOT (MATCH •OBJECT/R))

M22 " MATCH VAL " (TRANSF /C.2) (EVAL •DIFFR/R)
  •>
 (TRANSF /R.2) (NOT (TRANSF /C.2)) (NOT (EVAL •DIFFR/R))

M23 " MATCH FIN " (TRANSF /H.3)
  •>
 (MATCH /DA) (TRANSF /A.3) (NOT (TRANSF /H.3))

M23E " ERASE M.O " (MATCH /DA) (MATCH •OBJECT/A)
  •>
 (NOT (MATCH /DA)) (NOT (MATCH •OBJECT/A))

M24 " COMP DIFFIC " (TRANSF /A.3) (TRANSF /R.2) (NOT ((TRANSF /R.2)))
 (NOT ((TRANSF /R.2)))
  •>
 (TRANSF /DR.2) (GOAL •REDUCE•NEXT:TRANS/W) (NOT (TRANSF /A.3))
 (NOT (TRANSF /R.2))

M24E " ERS MVAL " (TRANSF /DR.2) (TRANSF /R.2) (NOT ((TRANSF /R.2)))
  •>
 (NOT (TRANSF /DR.2)) (NOT (TRANSF /R.2))

M24F " ERS MVAL- " (TRANSF /DR.2) (NOT ((TRANSF /R.2)))
  •>
 (NOT (TRANSF /DR.2))

M24N " ERS MVAL SV- " (TRANSF /DR.2) (TRANSF /R.2) (NOT ((TRANSF •RETRY/T)))
  •>
 (TRANSF /DR.2) (NOT (TRANSF /R.2))

M24S " ERS MVAL SV " (TRANSF /DR.2) (TRANSF /R.2) (TRANSF •RETRY/T)
  •>
 (TRANSF /DR.2) (GOAL •ALTDIFFR/T) (NOT (TRANSF /R.2))

M25 " SUC DESCR " (TRANSF /A.3) (NOT ((TRANSF /R.2))) (GOAL •DESOBJECT/T)
 (OBJECT •TYPEDESCRIBED/T) (GOAL •OBJECT/T)
  •>
 (GOAL /T) (NOT (TRANSF /A.3))

M26 " NEW REDUCE " (GOAL •REDUCE•NEXT:TRANS/W) (GOAL •DESOBJECT/T)
 (GOAL •OBJECT/T) (NOT ((GOAL •DIFFIC/T)))
  •>
 (FILE •GOAL/A) (EVAL •GOAL/A) (GOAL •SUPER/T) (GOAL •DIFFIC/T)
 (GOAL •TYPEREDUCE/T) (GOAL •OBJECT/T) (GOAL •DIFFR/T) (GOAL •NEXT:TRANS/C)
 (NOT (GOAL •REDUCE•NEXT:TRANS/W))

M27 " NEW REDUCE " (GOAL •REDUCE•NEXT:TRANS/W) (GOAL •DESOBJECT/T)
 (GOAL •OBJECT/T) (GOAL •DIFFIC/T)
  •>
 (FILE •GOAL/A) (EVAL •GOAL/A) (GOAL •SUPER/T) (GOAL •DIFFIC/T)
 (GOAL •TYPEREDUCE/T) (GOAL •OBJECT/T) (GOAL •DIFFR/T) (GOAL •NEXT:TRANS/C)
 (NOT (GOAL •REDUCE•NEXT:TRANS/W))

M30 " REDUCE:G " (REDUCE /A) (GOAL •DIFFR/T) (NOT (GOAL •RETRY/T))
  •>
 (APPLY •LOC:PROG/A) (REDUCE /A.1) (NOT (REDUCE /A))

M31 " SEL OP " (REDUCE /A.1) (APPLY •LOC:PROG/R) (NOT (SET •TYPESET/T))
 (MOVE:OPR •TYPE/T)
  •>
 (GENRT •DESASG/A) (GOAL •OPR/T) (NOT (REDUCE /A.1)) (NOT (APPLY •LOC:PROG/R))

M32 " SEL OP SET " (REDUCE /A.1) (APPLY •LOC:PROG/R) (SET •TYPESET/T)
 (MOVE:OPR •SET/T) (MOVE:OPR •TYPE/T)
  •>
 (GENRT •DESASG/A) (GOAL •OPR/T) (NOT (REDUCE /A.1)) (NOT (APPLY •LOC:PROG/R))

M33 " SEL OP FORM " (REDUCE /A.1) (APPLY •LOC:PROG/R) (NOT (SET •TYPESET/T))
 (FORM:OPR •TYPE/T) (GOAL •OBJECT/T)
  •>
 (APPLY •FORM:OPR/A) (REDUCE •FORM/C) (NOT (REDUCE /A.1))
 (NOT (APPLY •LOC:PROG/R))

M33S " SEL FORM SET " (REDUCE /A.1) (APPLY •LOC:PROG/R) (SET •TYPESET/T)
 (MOVE:OPR •SET/T) (FORM:OPR •TYPE/T) (GOAL •OBJECT/T)
  •>
 (APPLY •FORM:OPR/A) (REDUCE •FORM/C) (NOT (REDUCE /A.1))
 (NOT (APPLY •LOC:PROG/R))

M34 " SEL DES ASG " (GENRT •DESASG/A) (MOVE:OPR •COMPON/T)
  •>
 (GENRT •DESASG/A.1) (APPLY •MOVE:OPR/WA) (GOAL •DESASG/T)
 (NOT (GENRT •DESASG/A))

M34A " SEL DES ASG ARB " (GENRT •DESASG/A) (NOT ((MOVE:OPR •COMPON/T)))
 (MOVE:OPR •COMPON/T) (COMPON •VARBL/T) (MOVE:OPR •VALCHANGE/T)
 (VARBL •DOMAIN/T)
  •>
 (LOC:PROG •LINK/DT) (LOC:PROG •LINK/T) (FILE •DESASG/A) (GENRT •FEASASG/A)
 (VARBL •ASG/M.1) (APPLY •MOVE:OPR/WA) (GOAL •DESASG/T) (NOT (GENRT •DESASG/A))

M34M " SEL DES ASG• " (GENRT •DESASG/A) (MOVE:OPR •COMPON/T)
  •>
 (GENRT •DESASG/A.1) (APPLY •MOVE:OPR/WA) (GOAL •DESASG/T)
 (NOT (GENRT •DESASG/A))

M34N " SEL DES ASG• " (GENRT •DESASG/A) (MOVE:OPR •COMPON/T)
  •>
 (GENRT •DESASG/A.1) (APPLY •MOVE:OPR/WA) (GOAL •DESASG/T)
 (NOT (GENRT •DESASG/A))

M34R " ERS SEL " (GENRT •DESASG/A) (NOT ((MOVE:OPR •COMPON/T)))
 (NOT
  ((MOVE:OPR •COMPON/T) (COMPON •VARBL/T) (MOVE:OPR •VALCHANGE/T)
     (VARBL •DOMAIN/T)))
  •>
 (GENRT •DESASG/E) (NOT (GENRT •DESASG/A))

M35 " GET COMP• " (GENRT •DESASG/A.1) (LOC:PROG •LINK/T)
 (NOT ((LOC:PROG •LINK/T)))
  •>
 (GENRT •DESASG/A.2) (NOT (GENRT •DESASG/A.1))

M35G " GET COMP " (GENRT •DESASG/A.1) (NOT ((LOC:PROG •LINK/T)))
  •>
 (LOC:PROG •LINK/W) (GENRT •DESASG/A.2) (NOT (GENRT •DESASG/A.1))

M36 " GEN DES ASG " (GENRT •DESASG/A.2) (COMPON •VARBL/T) (LOC:PROG •LINK/T)
 (VARBL •DOMAIN/T)
  •>
 (GENRT •DESASG/A.3) (FILE •DESASG/A) (GENRT •FEASASG/A) (VARBL •ASG/M.1)

M36A " GEN DES ASG ARB " (GENRT •DESASG/A.2) (COMPON •VARBL/T)
 (MOVE:OPR •VALCHANGE/T) (MOVE:OPR •COMPON/T) (VARBL •DOMAIN/T)
  •>
 (LOC:PROG •LINK/DT) (FILE •DESASG/A) (GENRT •FEASASG/A) (VARBL •ASG/M.1)
 (NOT (GENRT •DESASG/A.2))

M37 " ERS LC " (LOC:PROG •LINK/DT) (LOC:PROG •LINK/T)
 (NOT ((GENRT •DESASG/A.2))) (NOT ((GENRT •DESASG/A)))
  •>
 (NOT (LOC:PROG *LINK/DT)) (NOT (LOC:PROG *LINK/T))

M37R " ERS LC RE-AS " (LOC:PROG *LINK/DT) (GENRT *DESASG/A.2)
   *>
 (GENRT *DESASG/A.2) (NOT (LOC:PROG *LINK/DT))

M37S " ERS LC RE-AS " (LOC:PROG *LINK/DT) (GENRT *DESASG/A)
   *>
 (GENRT *DESASG/A) (NOT (LOC:PROG *LINK/DT))

M38 " GEN DES ASG* " (GENRT *DESASG/A.3) (GENRT *DESASG/A.2) (COMPON *VARBL/T)
 (VARBL *LINK/T) (LOC:PROG *LINK/T) (VARBL *DOMAIN/T)
   *>
 (LOC:PROG *LINK/DT) (FILE *DESASG/A) (GENRT *FEASASG/A) (VARBL *ASG/M.1)
 (NOT (GENRT *DESASG/A.3)) (NOT (GENRT *DESASG/A.2))

M38F " GEN ASG* - " (GENRT *DESASG/A.3) (GENRT *DESASG/A.2) (COMPON *VARBL/T)
 (VARBL *LINK/T) (NOT (LOC:PROG *LINK/T)) (GOAL *DESASG/T) (VARBL *ASG/M.1)
 (GENRT *FEASASG/A) (FILE *DESASG/A)
   *>
 (LOC:PROG *LINK/DT) (NOT (GENRT *DESASG/A.3)) (NOT (GENRT *DESASG/A.2))
 (NOT (GOAL *DESASG/T)) (NOT (VARBL *ASG/M.1)) (NOT (GENRT *FEASASG/A))
 (NOT (FILE *DESASG/A))

M39 " GEN DES ASG*- " (GENRT *DESASG/A.3) (GENRT *DESASG/A.2)
 (NOT ((COMPON *VARBL/T) (VARBL *LINK/T)))
   *>
 (LOC:PROG *LINK/DT) (NOT (GENRT *DESASG/A.3)) (NOT (GENRT *DESASG/A.2))

M40 " TRY APPLY " (APPLY *MOVE:OPR/WA) (GOAL *NEWFEAS/T)
 (NOT ((APPLY *MOVE:OPR/WA) (GOAL *NEWFEAS/T))) (NOT ((GOAL *NEWFEAS/T)))
 (NOT ((GOAL *NEWFEAS/T))) (GOAL *OBJECT/T) (NOT ((GENRT *FEASASG/A)))
   *>
 (APPLY *MOVE:OPR/A) (APPLY *MOVE:OPR/C) (NOT (APPLY *MOVE:OPR/WA))
 (NOT (GOAL *NEWFEAS/T))

M40H " TRY APPLY MULT " (APPLY *MOVE:OPR/WA) (GOAL *NEWFEAS/T)
 (NOT ((APPLY *MOVE:OPR/WA) (GOAL *NEWFEAS/T)))
   *>
 (APPLY *MOVE:OPR/WA) (APPLY *MOVE:OPR/WH)

M40R " RE-AS FEASASG " (GOAL *NEWFEAS/T) (GENRT *FEASASG/A) (APPLY *MOVE:OPR/WA)
   *>
 (GENRT *FEASASG/A) (APPLY *MOVE:OPR/WH) (NOT (APPLY *MOVE:OPR/WA))

M40U " UN-HOLD TRYAPP " (APPLY *MOVE:OPR/WH)
   *>
 (APPLY *MOVE:OPR/WA) (NOT (APPLY *MOVE:OPR/WH))

M41 " APPLY SUC " (APPLY *MOVE:OPR/C) (APPLY *MOVE:OPR/R)
   *>
 (FILE *OBJECT/A) (GOAL /T) (NOT (APPLY *MOVE:OPR/C)) (NOT (APPLY *MOVE:OPR/R))
 (NOT (APPLY *MOVE:OPR/WA.A))

M42 " EVAL OP DIFFR " (APPLY *MOVE:OPR/C) (APPLY *MOVE:OPR/E)
   *>
 (EVAL *DIFFR/A) (APPLY *MOVE:OPR/WC.3) (APPLY *MOVE:OPR/WA)
 (NOT (APPLY *MOVE:OPR/C)) (NOT (APPLY *MOVE:OPR/E))

M43 " DIFFR DIFFIC+ " (APPLY *MOVE:OPR/WC.3) (EVAL *DIFFR/R)
 (APPLY *MOVE:OPR/WM.3) (NOT ((APPLY *MOVE:OPR/WC.3) (EVAL *DIFFR/R)))
   *>
 (APPLY *MOVE:OPR/WA.A) (APPLY *MOVE:OPR/WM.3) (NOT (APPLY *MOVE:OPR/WC.3))
 (NOT (EVAL *DIFFR/R))

M43F " DIFFR DIFFIC1 " (APPLY *MOVE:OPR/WC.3) (EVAL *DIFFR/R)
 (NOT ((APPLY *MOVE:OPR/WM.3))) (NOT ((APPLY *MOVE:OPR/WC.3) (EVAL *DIFFR/R)))
   *>
 (APPLY *MOVE:OPR/WA.A) (APPLY *MOVE:OPR/WM.3) (NOT (APPLY *MOVE:OPR/WC.3))
 (NOT (EVAL *DIFFR/R))

M43L " DIFFR DIFFIC- " (APPLY *MOVE:OPR/WC.3) (EVAL *DIFFR/R)
 (APPLY *MOVE:OPR/WM.3)
   *>
 (APPLY *MOVE:OPR/WA.A) (APPLY *MOVE:OPR/WM.3) (NOT (APPLY *MOVE:OPR/WC.3))
 (NOT (EVAL *DIFFR/R))

M44 " APPLY FAIL " (APPLY *MOVE:OPR/C) (NOT ((APPLY *MOVE:OPR/R)))
 (NOT ((APPLY *MOVE:OPR/F)))
   *>
 (APPLY *MOVE:OPR/DA) (APPLY *MOVE:OPR/WA) (APPLY *MOVE:OPR/WA.A)
 (NOT (APPLY *MOVE:OPR/C))

M44E " APPLY ERASE " (APPLY *MOVE:OPR/DA) (APPLY *MOVE:OPR/A) (VARBL *ASG/T)
   *>
 (NOT (APPLY *MOVE:OPR/DA)) (NOT (APPLY *MOVE:OPR/A)) (NOT (VARBL *ASG/T))

M45 " SEL DIFFR " (APPLY *MOVE:OPR/WA.A) (APPLY *MOVE:OPR/WM.3)
 (NOT ((GOAL *NEWFEAS/T))) (NOT ((APPLY *LOC:PROG/A)))
 (NOT ((APPLY *MOVE:OPR/WM.3))) (NOT ((APPLY *MOVE:OPR/WM.3)))
 (NOT ((APPLY *MOVE:OPR/WM.3)))
   *>
 (APPLY *MOVE:OPR/WE) (NOT (APPLY *MOVE:OPR/WA.A)) (NOT (APPLY *MOVE:OPR/WM.3))
 (NOT (APPLY *MOVE:OPR/WA))

M451 " SEL IDENT " (APPLY *MOVE:OPR/WA.A) (APPLY *MOVE:OPR/WM.3)
 (NOT ((GOAL *NEWFEAS/T))) (NOT ((APPLY *LOC:PROG/A)))
 (NOT ((APPLY *MOVE:OPR/WM.3)))
   *>
 (APPLY *MOVE:OPR/WM.3)

M46 " NO DIFFR " (APPLY *MOVE:OPR/WA.A) (NOT ((APPLY *MOVE:OPR/WM.3)))
 (NOT ((APPLY *MOVE:OPR/WA)))
   *>
 (GOAL /F) (GOAL *EXHAUST/T) (NOT (APPLY *MOVE:OPR/WA.A))

M46T " NO DIFFR TA. " (APPLY *MOVE:OPR/WA.A) (NOT ((APPLY *MOVE:OPR/WM.3)))
 (APPLY *MOVE:OPR/WA) (NOT ((GOAL *NEWFEAS/T)))
   *>
 (GOAL /F) (GOAL *EXHAUST/T) (NOT (APPLY *MOVE:OPR/WA.A))
 (NOT (APPLY *MOVE:OPR/WA))

M47 " TRYAPP RES " (APPLY *MOVE:OPR/WE) (GOAL *TYPEREDUCE/T) (GOAL *OBJECT/T)
 (GOAL *DIFFIC/T)
   *>
 (FILE *GOAL/A) (EVAL *GOAL/A) (GOAL *TYPEAPPLY/T) (GOAL *DESASG/T)
 (GOAL *OPR:DIFFR) (GOAL *OPR/T) (GOAL *OBJECT/T) (GOAL *DIFFIC/T)
 (GOAL *SUPER/T) (NOT (APPLY *MOVE:OPR/WE))

M48 " RETRY ASG " (REDUCE /A) (GOAL *NEWFEAS/T) (GOAL *RETRY/T)
   *>
 (APPLY *MOVE:OPR/WA) (NOT (REDUCE /A)) (NOT (GOAL *RETRY/T))

M49 " RETRY OPD " (REDUCE /A) (GOAL *RETRY/T) (NOT ((GOAL *NEWFEAS/T)))
 (GOAL *DESASG/T)
   *>
 (APPLY *MOVE:OPR/WA.A) (NOT (REDUCE /A)) (NOT (GOAL *RETRY/T))

M49N " RETRY NOTHING " (REDUCE /A) (GOAL *RETRY/T) (NOT ((GOAL *DESASG/T)))
   *>
 (REDUCE /A) (NOT (GOAL *RETRY/T))

M50 " MOVE OP " (MOVE:OPR *METHOD/A) (NOT (GOAL *RETRY/T)) (GOAL *OPR:DIFFR)
 (GOAL *DIFFIC/T)
   *>
 (GOAL *REDUCE*NEXT:APPLY/W) (NOT (MOVE:OPR *METHOD/A))

M51 " NEW RED APP " (GOAL *REDUCE*NEXT:APPLY/W) (GOAL *DESASG/T)
 (GOAL *DIFFIC/T) (GOAL *OBJECT/T)
   *>
 (FILE *GOAL/A) (EVAL *GOAL/A) (GOAL *SUPER/T) (GOAL *DIFFIC/T)
 (GOAL *TYPEREDUCE/T) (GOAL *DIFFR/T) (GOAL *OBJECT/T) (GOAL *NEXT:APPLY/C)
 (NOT (GOAL *REDUCE*NEXT:APPLY/W))

[illegible] " MOVE OP - DIF " (MOVE:OPR *METHOD/A) (NOT (GOAL *RETRY/T)) (GOAL *OPR/T)
 (NOT ((GOAL *OPR:DIFFR))) (GOAL *DESASG/T)
   *>
 (GENRT *FEASASG/A) (APPLY *MOVE:OPR/WA) (NOT (MOVE:OPR *METHOD/A))

M57 " TRYAPP RES " (APPLY *MOVE:OPR/WE) (GOAL *TYPEAPPLY/T) (GOAL *DIFFIC/T)
   *>
 (GOAL *REDUCE*NEXT:APPLY/W) (NOT (APPLY *MOVE:OPR/WE)) (NOT (GOAL *DIFFIC/T))
 (GOAL *DIFFIC/T)

M58 " RETRY ASG " (MOVE:OPR *METHOD/A) (GOAL *NEWFEAS/T) (GOAL *RETRY/T)
   *>
 (APPLY *MOVE:OPR/WA) (NOT (MOVE:OPR *METHOD/A)) (NOT (GOAL *RETRY/T))

M59 " RETRY OPD " (MOVE:OPR *METHOD/A) (GOAL *RETRY/T) (NOT ((GOAL *NEWFEAS/T)))
   *>
 (APPLY *MOVE:OPR/WA.A) (NOT (MOVE:OPR *METHOD/A)) (NOT (GOAL *RETRY/T))

K0 " MATCHDIFF TOP " (MATCH *OBJECT/A) (OBJECT *TOPNODE/T)
   *>
 (MATCH *OBJECT/A)
```

```
K1 " MD * " (MATCH *OBJECT/A) (NODE *LINK/T) (NOT ((OBJECT *RESTR/T)))
   =>
 (MATCH *OBJECT/A)

K3 " MD BAD VAL " (MATCH *OBJECT/A) (NODE *VALUE/T)
   =>
 (MATCH *OBJECT/A.2) (NOT (MATCH *OBJECT/A))

K4 " MD UNDEF N1 " (MATCH *OBJECT/A) (NODE *LINK/T) (NOT ((NODE *LINK/T)))
   =>
 (MATCH *OBJECT/A)

K5 " MD UNDEF N2 " (MATCH *OBJECT/A) (NODE *LINK/T) (NOT ((NODE *LINK/T)))
   =>
 (MATCH *OBJECT/A)

K6 " MD UNDEF V1 " (MATCH *OBJECT/A) (NODE *VALUE/T) (NOT ((NODE *VALUE/T)))
   =>
 (MATCH *OBJECT/A.2) (NOT (MATCH *OBJECT/A))

K7 " MD UNDEF V2 " (MATCH *OBJECT/A) (NODE *VALUE/T) (NOT ((NODE *VALUE/T)))
   =>
 (MATCH *OBJECT/A.2) (NOT (MATCH *OBJECT/A))

K8 " LOC EXTR " (MATCH *OBJECT/A.2) (NODE *LINK/T) (NOT ((OBJECT *TOPNODE/T)))
   =>
 (MATCH *OBJECT/A.2) (DIFFR *LINK/T)

K9 " LOC EXTR TOP " (MATCH *OBJECT/A.2) (NODE *LINK/T) (OBJECT *TOPNODE/T)
 (NOT ((MATCH *OBJECT/I)))
   =>
 (FILE *LOC-PROG/A) (DIFFR *LINK/T) (MATCH *OBJECT/C.3)
 (NOT (MATCH *OBJECT/A.2))

K10 " MD RESULT F " (MATCH *OBJECT/C.3) (DIFFR *NAME/T)
   =>
 (MATCH *OBJECT/R) (NOT (MATCH *OBJECT/C.3)) (NOT (DIFFR *NAME/T))

K11 " LOC EXTR TOP I " (MATCH *OBJECT/A.2) (NODE *LINK/T) (OBJECT *TOPNODE/T)
 (MATCH *OBJECT/I)
   =>
 (MATCH *OBJECT/R.1) (DIFFR *LINK/T) (NOT (MATCH *OBJECT/A.2))

T1 " ADD LINK1 " (ADD *LINK/A) (OBJECT *TOPNODE/T) (NOT ((NODE *LINK/T)))
   =>
 (ADD *LINK/M) (NODE *LINK/T) (NOT (ADD *LINK/A))

T2 " ADD LINK 1- " (ADD *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
   =>
 (ADD *LINK/M) (NOT (ADD *LINK/A))

T3 " ADD LINK N " (ADD *LINK/A) (ADD *LINK/M) (NOT ((ADD *LINK/A)))
 (NOT ((NODE *LINK/T)))
   =>
 (ADD *LINK/M) (NODE *LINK/T) (NOT (ADD *LINK/A))

T4 " ADD LINK N- " (ADD *LINK/A) (ADD *LINK/M) (NOT ((ADD *LINK/A)))
 (NODE *LINK/T)
   =>
 (ADD *LINK/M) (NOT (ADD *LINK/A))

T5 " ADD LINK V " (ADD *LINK/A) (ADD *LINK/M) (NOT ((ADD *LINK/A)))
   =>
 (NODE *LINK/T) (NODE *VALUE/T) (NOT (ADD *LINK/A)) (NOT (ADD *LINK/M))

T6 " ADD LINK VT " (ADD *LINK/A) (NOT ((ADD *LINK/M))) (OBJECT *TOPNODE/T)
 (NOT ((ADD *LINK/A)))
   =>
 (NODE *LINK/T) (NODE *VALUE/T) (NOT (ADD *LINK/A))

T10 " REM LINK ALL TOP " (REMOVE *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
 (NOT ((REMOVE *LINK/A)))
   =>
 (REMOVE *LINK/A) (REMOVE *LINK/M) (NOT (NODE *LINK/T))

T11 " REM LINK SPEC TOP " (REMOVE *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
   =>
 (REMOVE *LINK/M) (NOT (REMOVE *LINK/A))

T12 " REM LINK ALL ARB " (REMOVE *LINK/A) (REMOVE *LINK/M) (NODE *LINK/T)
 (NOT ((REMOVE *LINK/A)))
   =>
 (REMOVE *LINK/A) (REMOVE *LINK/M) (NOT (NODE *LINK/T))

T13 " REM LINK VAL " (REMOVE *LINK/A) (REMOVE *LINK/M) (NODE *LINK/T)
 (NODE *VALUE/T)
   =>
 (NOT (REMOVE *LINK/A)) (NOT (REMOVE *LINK/M)) (NOT (NODE *LINK/T))
 (NOT (NODE *VALUE/T))

T14 " REM LINK ARB C " (REMOVE *LINK/A) (REMOVE *LINK/M) (NODE *LINK/T)
   =>
 (REMOVE *LINK/M) (NOT (REMOVE *LINK/A))

T20 " INCR LINK I " (INCR *LINK/I) (OBJECT *TOPNODE/T) (NODE *LINK/T)
   =>
 (INCR *LINK/M) (NOT (INCR *LINK/I))

T21 " INCR LINK N " (INCR *LINK/I) (INCR *LINK/M) (NOT ((INCR *LINK/I)))
 (NODE *LINK/T)
   =>
 (INCR *LINK/M) (NOT (INCR *LINK/I))

T22 " INCR LINK V " (INCR *LINK/I) (INCR *LINK/M) (NODE *LINK/T) (NODE *VALUE/T)
   =>
 (NOT (INCR *LINK/I)) (NOT (INCR *LINK/M)) (NOT (NODE *VALUE/T))
 (NODE *VALUE/T)

T23 " INCR LINK VT " (INCR *LINK/I) (OBJECT *TOPNODE/T) (NODE *LINK/T)
 (NODE *VALUE/T)
   =>
 (NOT (INCR *LINK/I)) (NOT (NODE *VALUE/T)) (NODE *VALUE/T)

T30 " DECR LINK I " (DECR *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
   =>
 (DECR *LINK/M) (NOT (DECR *LINK/A))

T31 " DECR LINK N " (DECR *LINK/A) (DECR *LINK/M) (NOT ((DECR *LINK/A)))
 (NODE *LINK/T)
   =>
 (DECR *LINK/M) (NOT (DECR *LINK/A))

T32 " DECR LINK V " (DECR *LINK/A) (DECR *LINK/M) (NODE *LINK/T) (NODE *VALUE/T)
   =>
 (NOT (DECR *LINK/A)) (NOT (DECR *LINK/M)) (NOT (NODE *VALUE/T))
 (NODE *VALUE/T)

T33 " DECR LINK VT " (DECR *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
 (NODE *VALUE/T)
   =>
 (NOT (DECR *LINK/A)) (NOT (NODE *VALUE/T)) (NODE *VALUE/T)

T60 " COPY LINK I " (COPY *LINK/A) (OBJECT *TOPNODE/T) (NODE *LINK/T)
   =>
 (COPY *LINK/M) (NOT (COPY *LINK/A))

T61 " COPY LINK N " (COPY *LINK/A) (COPY *LINK/M) (NOT ((COPY *LINK/A)))
 (NODE *LINK/T)
   =>
 (COPY *LINK/M) (NOT (COPY *LINK/A))

T62 " COPY LINK V " (COPY *LINK/A) (COPY *LINK/M) (NOT ((COPY *LINK/A)))
 (NODE *LINK/T) (NODE *VALUE/T)
   =>
 (NODE *VALUE/T) (NOT (COPY *LINK/A)) (NOT (COPY *LINK/M))

T63 " COPY LINK T " (COPY *LINK/A) (NOT ((COPY *LINK/M))) (NOT ((COPY *LINK/A)))
 (OBJECT *TOPNODE/T) (NODE *LINK/T) (NODE *VALUE/T)
   =>
 (NODE *VALUE/T) (NOT (COPY *LINK/A))

C1 " COPY OBJ TOP " (COPY *OBJECT/A) (OBJECT *TOPNODE/T)
   =>
 (COPY *OBJECT/A) (OBJECT *TOPNODE/T)

C2 " COPY OBJ N " (COPY *OBJECT/A) (NODE *LINK/T) (NOT ((NODE *VALUE/T)))
   =>
 (COPY *OBJECT/A) (NODE *LINK/T)

C3 " COPY OBJ NV " (COPY *OBJECT/A) (NODE *LINK/T) (NODE *VALUE/T)
   =>
 (NODE *LINK/T) (NODE *VALUE/T) (NOT (COPY *OBJECT/A))

C4 " COPY OBJ - " (COPY *OBJECT/A) (NOT ((NODE *LINK/T)))
 (NOT ((OBJECT *TOPNODE/T)))
   =>
 (NOT (COPY *OBJECT/A))
```

```
D1 " DIFFR EVAL " (EVAL •DIFFR/A)
   •>
 (APPLY •LOC$PROG/A) (EVAL •DIFFR/A.1) (NOT (EVAL •DIFFR/A))

D2 " DIFFR EVAL R1 " (EVAL •DIFFR/A.1) (APPLY •LOC$PROG/R)
   •>
 (EVAL •DIFFR/A.2) (NOT (EVAL •DIFFR/A.1)) (NOT (APPLY •LOC$PROG/R))

D3 " DIFFR EVAL R2 " (EVAL •DIFFR/A.2)
   •>
 (EVAL •DIFFR/R) (NOT (EVAL •DIFFR/A.2))

D4 " DIFFR EVAL R2• " (EVAL •DIFFR/A.2)
   •>
 (EVAL •DIFFR/R) (NOT (EVAL •DIFFR/A.2))

D5 " DIFFR EVAL R2T " (EVAL •DIFFR/A.2)
   •>
 (EVAL •DIFFR/R) (NOT (EVAL •DIFFR/A.2))

V1 " TRANSF " (TRACE •GOAL/A) (GOAL •TYPETRANSFORM/T) (GOAL •SUPER/T)
 (GOAL •TRACE£EVEL/T) (GOAL •OBJECT/T) (TRACE •INDENT/T) (GOAL •DESOBJECT/T)
 (NOT ((GOAL •ANTEC/T)))
   •>
 (TRACE /T) (TRACE •OBJECT/A) (GOAL •TRACE£EVEL/T) (NOT (TRACE •GOAL/A))
 (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

V2 " OBJ " (TRACE •OBJECT/A) (OBJECT •EXTREPR/T) (TRACE •INDENT/T)
   •>
 (TRACE /T) (NOT (TRACE •OBJECT/A))

V3 " TRANSF " (TRACE •GOAL/A) (GOAL •TYPETRANSFORM/T) (GOAL •SUPER/T)
 (GOAL •TRACE£EVEL/T) (GOAL •OBJECT/T) (GOAL •DESOBJECT/T) (GOAL •ANTEC/T)
 (TRACE •INDENT/T)
   •>
 (TRACE /T) (TRACE •OBJECT/A) (GOAL •TRACE£EVEL/T) (NOT (TRACE •GOAL/A))
 (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

V4 " APPLY " (TRACE •GOAL/A) (GOAL •TYPEAPPLY/T) (GOAL •SUPER/T)
 (GOAL •TRACE£EVEL/T) (GOAL •OBJECT/T) (GOAL •DIFFIC/T) (GOAL •DESASG/T)
 (GOAL •OPR/T) (NOT ((GOAL •ANTEC/T))) (TRACE •INDENT/T)
   •>
 (TRACE /T) (TRACE •DESASG/A) (TRACE •OBJECT/A) (GOAL •TRACE£EVEL/T)
 (NOT (TRACE •GOAL/A)) (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

V5 " APPLY " (TRACE •GOAL/A) (GOAL •TYPEAPPLY/T) (GOAL •SUPER/T)
 (GOAL •TRACE£EVEL/T) (GOAL •OBJECT/T) (GOAL •DIFFIC/T) (GOAL •DESASG/T)
 (GOAL •OPR/T) (GOAL •ANTEC/T) (TRACE •INDENT/T)
   •>
 (TRACE /T) (TRACE •DESASG/A) (TRACE •OBJECT/A) (GOAL •TRACE£EVEL/T)
 (NOT (TRACE •GOAL/A)) (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

V6 " REDUCE " (TRACE •GOAL/A) (GOAL •TYPEREDUCE/T) (GOAL •SUPER/T)
 (GOAL •TRACE£EVEL/T) (GOAL •OBJECT/T) (GOAL •DIFFR/T) (GOAL •DIFFIC/T)
 (TRACE •INDENT/T) (OBJECT •EXTREPR/T)
   •>
 (TRACE /T) (TRACE •OBJECT/A) (GOAL •TRACE£EVEL/T) (NOT (TRACE •GOAL/A))
 (NOT (TRACE •INDENT/T)) (TRACE •INDENT/T)

V7 " ASG 1 " (TRACE •DESASG/A) (VARBL •ASG/M.2) (NOT ((VARBL •ASG/M.2)))
 (TRACE •INDENT/T)
   •>
 (TRACE /T) (NOT (TRACE •DESASG/A))

V8 " ASG 2 " (TRACE •DESASG/A) (VARBL •ASG/M.2) (NOT ((VARBL •ASG/M.2)))
 (TRACE •INDENT/T)
   •>
 (TRACE /T) (NOT (TRACE •DESASG/A))

V9 " ASG 3 " (TRACE •DESASG/A) (VARBL •ASG/M.2) (TRACE •INDENT/T)
   •>
 (TRACE /T) (NOT (TRACE •DESASG/A))

X1 " EXT REPR " (OBJECT •EXTREPR/G) (OBJECT •TOPNODE/T) (NODE •LINK/T)
   •>
 (NODE •EXTREPR/G) (NODE •EXTREPR/T) (NOT (OBJECT •EXTREPR/G))

X2 " DESC " (NODE •EXTREPR/G) (NODE •LINK/T)
   •>
 (NODE •EXTREPR/G) (NODE •EXTREPR/T)

X3 " BOTT " (NODE •EXTREPR/G) (NODE •VALUE/T) (NODE •EXTREPR/T)
   •>
 (NODE •EXTREPR/T) (NOT (NODE •EXTREPR/G))

X4 " BOTT NIL " (NODE •EXTREPR/G) (NOT ((NODE •VALUE/T))) (NOT ((NODE •LINK/T)))
 (NODE •EXTREPR/T)
   •>
 (NODE •EXTREPR/T) (NOT (NODE •EXTREPR/G))

X5 " ASC " (NODE •EXTREPR/G) (NODE •EXTREPR/T) (NOT ((NODE •EXTREPR/G)))
   •>
 (NODE •EXTREPR/T) (NOT (NODE •EXTREPR/G))

X6 " TOP " (NODE •EXTREPR/T) (OBJECT •TOPNODE/T) (NOT ((NODE •EXTREPR/G)))
 (TRACE •INDENT/T)
   •>
 (OBJECT •EXTREPR/T) (NOT (NODE •EXTREPR/T))
```

Appendix C. CROSS-REFERENCE OF FIRST ABSTRACTION

ADD
LHSUSES T1 T2 T3 T4 T5 T6
NESTEDL T3 T4 T5 T6
RHSUSES T1 -T1 T2 -T2 T3 -T3 T4 -T4 -T5 -T6

APPLY
LHSUSES M31 M32 M33 M33S M40 M40H M40R M40U M41 M42 M43 M43F M43L M44 M44E
M45 M45I M46 M46T M47 M57 D2
NESTEDL M40 M40H M43 M43F M44 M45 M45I M46 M46T
RHSUSES M30 -M31 -M32 M33 -M33 M33S -M33S M34 M34A M34M M34N M40 -M40 M40H
M40R -M40R M40U -M40U -M41 M42 -M42 M43 -M43 M43F -M43F M43L -M43L M44 -M44
-M44E M45 -M45 M45I -M46 -M46T -M47 M48 M49 M52 -M57 M58 M59 D1 -D2

CHANGEPM
RHSUSES F5 F13 F34 F35 F36 F38 F56

COMPON
LHSUSES M34A M36 M36A M38 M38F
NESTEDL M34R M39

COPY
LHSUSES T40 T41 T42 T43 C1 C2 C3 C4
NESTEDL T41 T42 T43
RHSUSES T40 -T40 T41 -T41 -T42 -T43 C1 C2 -C3 -C4

DECR
LHSUSES T30 T31 T32 T33
NESTEDL T31
RHSUSES T30 -T30 T31 -T31 -T32 -T33

DIFFR
LHSUSES F1 F3 F4 F26 F27 F30 F32 K10
NESTEDL F1 F3 F4 F5 F27 F30 F32 F34 F35
RHSUSES F1 -F3 -F4 F5 F22 F24 -F26 -F27 -F30 -F32 K8 K9 -K10 K11

EVAL
LHSUSES E1 E2 E2R E3 E4 E8 M22 M43 M43F M43L D1 D2 D3 D4 D5
NESTEDL M43 M43F
RHSUSES -E1 -E2 -E2R -E3 -E4 -E8 E10 E11 E36 M21 -M22 M26 M27 M42 -M43 -M43F
-M43L M47 M51 D1 -D1 D2 -D2 D3 -D3 D4 -D4 D5 -D5

FILE
LHSUSES -E2 -E2R -E3 -E4 F1 F2 F3 F6 F7 F7N F8 F8N F8S F8T F8Y F8Z F9 F9N F10
F11 F13 F14 F15 F17 F20 F30 F36 F38 F40 F42 F42D F42L F42R F42S F42U F50 F51
F51M F51N F52 F53 M38F
NESTEDL F8S F8T F42 F51N
RHSUSES E10 E11 E36 F1 -F1 F2 -F2 -F3 F6 -F6 -F7 -F7N -F8 F8N -F8N F8S -F8S
F8T -F8T F8Y -F8Y -F8Z -F9 -F9N F10 -F10 F11 -F11 F13 -F13 F14 -F14 F15 -F15
F17 -F17 F20 -F20 -F30 F34 F35 -F36 -F38 -F40 F41 -F42 F42D -F42D F42L -F42L
F42R -F42R F42S -F42S F42U -F42U F50 -F50 F51 -F51 F51M -F51M -F51N F52 -F52
-F53 M26 M27 M34A M36 M36A M38 -M38F M41 M47 M51 K9

FORM2INPUTOPR
LHSUSES -M4 M5
RHSUSES M5

FORMOPR
LHSUSES M4 M5 M33 M33S
RHSUSES M4

GENRT
LHSUSES F51 M34 M34A M34M M34N M34R M35 M35G M36 M36A M37R M37S M38 M38F M39
M40R
NESTEDL M37 M40
RHSUSES F51 M31 M32 M34 -M34 M34A -M34A M34M -M34M M34N -M34N M34R -M34R M35
-M35 M35G -M35G M36 M36A -M36A M37R M37S M38 -M38 -M38F -M39 M40R M52

GOAL
LHSUSES E2 E2R E3 E4 E8 E10 E11 E12 E20 E21 E22 -E22 E23 -E23 E23R E24 E25
E26 E30 -E30 E31 E35 -E35 E36 F7 F7N F8 F8N F8T F9 F9N F41 F51 M1 M2 M3 M4 M5
M20 M20S M25 M26 M27 M30 -M30 M33 M33S M38F M40 M40H M40R M47 M48 M49 M49N
M50 -M50 M51 M52 -M52 M57 M58 M59 V1 V3 V4 V5 V6
NESTEDL E1 E2 E4 E12 E21 E26 E30 -E30 E35 E36 F7 F7N F8 F8S F8T F9 F9N M26
M40 M40H M45 M45I M46T M49 M49N M52 M59 V1 V4
RHSUSES E0 E2 -E2R E3 E4 -E8 E10 -E10 E11 -E11 E12 E20 -E20 E21 -E21 E22 -E22
E23 -E23 -E23R -E24 -E25 -E26 E31 E36 F7N F8S F8T F9N -F41 F42 F51 M20S M24
M24S M25 M26 -M26 M27 -M27 M31 M32 M34 M34A M34M M34N -M38F -M40 M41 M46 M46T
M47 -M48 -M49 -M49N M50 M51 -M51 M57 -M57 -M58 -M59 V1 V3 V4 V5 V6

GPSR
LHSUSES E0
RHSUSES -E0

INCR
LHSUSES T20 T21 T22 T23
NESTEDL T21
RHSUSES T20 -T20 T21 -T21 -T22 -T23

LOCPROG
LHSUSES M35 M36 M37 M37R M37S M38 -M38F
NESTEDL M35 M35G
RHSUSES F3 F4 M34A M35G M36A -M37 -M37R -M37S M38 M38F M39

MATCH
LHSUSES F20 F21 F22 F23 F24 F25 F28 F42D F42L F42R M21 M23E K0 K1 K3 K4 K5 K6
K7 K8 K9 K10 K11
NESTEDL F20 F23 F25 F28 F40 F42 F42D F42L K9
RHSUSES F13 F15 F17 F20 -F20 F21 -F21 -F22 -F23 -F24 -F25 -F28 F40 F42D F42L
F42R M20 -M21 M23 -M23E K0 K1 K3 -K3 K4 K5 K6 -K6 K7 -K7 K8 K9 -K9 K10 -K10
K11 -K11

MOVEOPR
LHSUSES M3 M31 M32 M33S M34 M34A M34M M34N M36A M50 M52 M58 M59
NESTEDL M34A M34R
RHSUSES M3 -M50 -M52 -M58 -M59

NETP
LHSUSES F4 F5 F5E F13 F19 F30 F32 F34 F35 F40 F55 F56 F57
NESTEDL F34
RHSUSES E0 F3 F4 F5 -F5 F5E F10 -F11 F13 F19 F30 F32 F34 -F34 -F35 -F40 F53
F55 F56 -F56 F57

NODE
LHSUSES F46 F47 K1 K3 K4 K5 K6 K7 K8 K9 K11 T2 T4 T10 T11 T12 T13 T14 T20 T21
T22 T23 T30 T31 T32 T33 T40 T41 T42 T43 C2 C3 X1 X2 X3 X4 X5 X6
NESTEDL F46 F48 K4 K5 K6 K7 T1 T3 C2 C4 X4 X5 X6
RHSUSES -F46 -F47 T1 T3 T5 T6 -T10 -T12 -T13 T22 -T22 T23 -T23 T32 -T32 T33
-T33 T42 T43 C2 C3 X1 X2 X3 -X3 X4 -X4 X5 -X5 -X6

OBJECT
LHSUSES F13 F14 F17 -F34 F35 F41 F44 F46 F47 F48 M25 K0 K9 K11 T1 T2 T6 T10
T11 T20 T23 T30 T33 T40 T43 C1 V2 V8 X1 X6
NESTEDL F13 F15 F48 K1 K8 C4
RHSUSES E0 F5 F10 -F14 F40 F42 F44 -F44 F46 -F47 -F48 C1 -X1 X6

REDUCE
LHSUSES M30 M31 M32 M33 M33S M48 M49 M49N
RHSUSES M2 M30 -M30 -M31 -M32 M33 -M33 M33S -M33S -M48 -M49 M49N

REMOVE
LHSUSES T10 T11 T12 T13 T14
NESTEDL T10 T12
RHSUSES T10 T11 -T11 T12 -T13 T14 -T14

SELECT
LHSUSES E30 E31 E32 E35 E36 E37 M1 M2 M3 M4 M5
NESTEDL E30 E31 E36
RHSUSES E1 E2R E8 E22 E23R E24 E26 E30 -E30 E31 -E32 E35 -E35 E36 -E37 -M1
-M2 -M3 -M4 -M5

SET
LHSUSES -M31 M32 -M33 M33S

TRACE
LHSUSES E23R E31 E36 F5 F40 V1 V2 V3 V4 V5 V6 V7 V8 V9 X6
RHSUSES E0 E2 E2R E3 E4 E8 E10 E11 E12 E20 E21 E22 E23R E24 E25 E26 E31 -E31
E36 -E36 F5 F6 F40 V1 -V1 V2 -V2 V3 -V3 V4 -V4 V5 -V5 V6 -V6 V7 -V7 V8 -V8 V9
-V9

TRANSF
LHSUSES E25 E26 M20 M20S M22 M23 M24 M24E M24F M24N M24S M25
NESTEDL E24 M24 M24E M24F M24N M25
RHSUSES E25 M1 M20 -M20 -M20S M21 M22 -M22 M23 -M23 M24 -M24 -M24E -M24F M24N
-M24N M24S -M24S -M25

VARBL
LHSUSES F50 F51 F53 F55 M34A M36 M36A M38 M38F M44E V7 V8 V9
NESTEDL F50 F53 F55 F56 M34R M39 V7 V8
RHSUSES F50 -F51 F53 -F53 F55 -F55 M34A M36 M36A M38 -M38F -M44E

Appendix D. SECOND ABSTRACTION

```
E0 " PROB INIT "        GPSR .> OBJECT TRACE NETP GOAL - GPSR

      % MODULE EVAL %

E1 " GOAL EVAL OK "      EVAL - GOAL .> SELECT - EVAL
E2 " GOAL EVAL- "       EVAL - FILE GOAL .> TRACE GOAL - EVAL
E2R " GOAL EVAL-R "      EVAL - FILE GOAL .> TRACE SELECT - EVAL - GOAL
E3 " GOAL EVAL-A "       EVAL - FILE GOAL .> TRACE GOAL - EVAL
E4 " GOAL EVAL-S "       EVAL - FILE GOAL .> TRACE GOAL - EVAL
E8 " SAME REP "        EVAL GOAL .> TRACE SELECT - EVAL - GOAL

E10 " SUC TRANS "       GOAL .> TRACE FILE EVAL GOAL
E11 " SUC APPLY "       GOAL .> TRACE FILE EVAL GOAL
E12 " SUC SUP "       GOAL .> TRACE GOAL
E20 " FAIL ANTEC "       GOAL .> TRACE GOAL
E21 " FAIL ANTEC- "      GOAL .> TRACE GOAL
E22 " CHECK RETRY. "      GOAL .> TRACE SELECT GOAL
E23 " CHECK RETRY EXH "          GOAL .> GOAL
E23R " FAIL REP "       GOAL TRACE .> TRACE SELECT - GOAL
E24 " CHECK RETRY TG "   GOAL - TRANSF .> TRACE SELECT - GOAL
E25 " RETRY TRANS "      GOAL TRANSF .> TRANSF TRACE - GOAL
E26 " RETRY TRANS- "     GOAL TRANSF .> TRACE SELECT - GOAL

      % MODULE SELECT %

E30 " TRY OLD GOALS "    SELECT GOAL .> SELECT
E31 " CHOOSE OLD "       SELECT GOAL TRACE .> SELECT TRACE GOAL
E32 " ERASE CH "       SELECT .> - SELECT
E35 " NEW OBJ CRIT "      SELECT GOAL .> SELECT
E36 " CHOOSE OBJ "       SELECT TRACE GOAL .> SELECT TRACE FILE EVAL GOAL
E37 " ERASE CH "       SELECT .> - SELECT

      % MODULE FILE %

F1 " FILE LOCPROG "     FILE DIFFR .> DIFFR FILE
F2 " EXTEND LP NET "     FILE .> FILE
F3 " ST LP NET COL "     FILE DIFFR .> NETP LOCPROG - FILE - DIFFR

F4 " COL LP NET "       NETP DIFFR .> NETP LOCPROG - DIFFR
F5 " COL LP NET D "      NETP - DIFFR TRACE .> NETP DIFFR CHANGEPM TRACE
  OBJECT
F5E " LAST MLT "       NETP .> NETP

F6 " FILE GOAL "        FILE .> TRACE FILE
F7 " REC GT- "   FILE GOAL .> - FILE
F7N " REC GT "   FILE GOAL .> GOAL - FILE
F8 " REC GA- "   FILE GOAL .> - FILE
F8N " REC GA "   FILE GOAL .> FILE
F8S " OLD NO DIFFR "     FILE - GOAL .> FILE GOAL
F8T " OLD DIFFR . "     FILE GOAL .> FILE GOAL
F8Y " ERS CSP "        FILE .> FILE
F8Z " ERS CS "   FILE .> - FILE
F9 " REC GR- "   FILE GOAL .> - FILE
F9N " REC GR "   FILE GOAL .> GOAL - FILE

F10 " FILE OBJECT "      FILE .> OBJECT NETP FILE
F11 " TEST FIN "       FILE .> FILE - NETP
F13 " NEW NET NETP "    FILE - OBJECT OBJECT NETP .> MATCH FILE CHANGEPM NETP
  NETP
F14 " SAME SET "       FILE OBJECT .> FILE - OBJECT
F15 " SAME OBJ "       FILE - OBJECT .> MATCH FILE
F17 " SAME EQV "       FILE OBJECT .> MATCH FILE
F19 " LAST MLT "       NETP .> NETP
F20 " OBJ DIFR "       FILE MATCH .> MATCH FILE

      % MODULE MATCH %

F21 " ERS MD1 "        MATCH .> MATCH
F22 " ERS MR1 "        MATCH .> DIFFR - MATCH
F23 " ERS MR1- "       MATCH .> - MATCH
F24 " ERS ML1 "        MATCH .> DIFFR - MATCH
F25 " ERS ML1- "       MATCH .> - MATCH
F26 " ERS MN1 "        DIFFR .> - DIFFR
F27 " ERS MN1- "       DIFFR .> - DIFFR
F28 " ERS MD! SIG "      MATCH .> - MATCH

      % MODULE FILE AGAIN %

F30 " EXT ON ST "       FILE NETP DIFFR .> NETP - FILE - DIFFR
F32 " EXT ON "   NETP DIFFR .> NETP - DIFFR
F34 " SPLIT ON P "      NETP - OBJECT - DIFFR NETP .> FILE CHANGEPM NETP -
  NETP
F35 " SPLIT ON P DM "   NETP OBJECT - DIFFR NETP .> FILE CHANGEPM - NETP -
  NETP
F36 " SPLIT OB1 "       FILE .> CHANGEPM - FILE
F38 " SPLIT OB2 "       FILE .> CHANGEPM - FILE

F40 " NO DIFFR "        FILE - MATCH NETP TRACE .> MATCH OBJECT TRACE - FILE -
  NETP
F41 " CHECK DONE "      OBJECT GOAL .> FILE - GOAL
F42 " ONET DONE "       FILE - MATCH .> OBJECT GOAL - FILE

F42D " ONET CONT D "    FILE MATCH .> MATCH FILE
F42L " ONET CONT L "    FILE MATCH .> MATCH FILE
F42R " ONET CONT R "    FILE MATCH .> MATCH FILE
F42S " ONET CONT S "    FILE .> FILE
F42U " UN-H OS "       FILE .> FILE

F44 " ERS OBJ "        OBJECT .> OBJECT
F46 " ERS OBJ N "       OBJECT NODE .> OBJECT - NODE
F47 " ERS OBJ NV "      OBJECT NODE .> - OBJECT - NODE
F48 " ERS OBJ N- "      OBJECT - NODE .> - OBJECT

F50 " FILE DES ASG "    FILE VARBL .> VARBL FILE
F51 " OLD DA "   VARBL FILE GOAL GENRT .> FILE GENRT GOAL - VARBL
F51M " MORE DA "       FILE .> FILE
F51N " MORE DA- "      FILE .> - FILE
F52 " EXTEND DA NET "    FILE .> FILE
F53 " DA NET COL "      FILE VARBL .> NETP VARBL - FILE

F55 " COL DA NET "      NETP VARBL .> NETP VARBL
F56 " COL DA NET D "     NETP - VARBL .> NETP CHANGEPM
F57 " LAST MLT "       NETP .> NETP

      % MODULE SELECT AGAIN %

M1 " SEL TF "   SELECT GOAL .> TRANSF - SELECT
M2 " SEL RED "   SELECT GOAL .> REDUCE - SELECT
M3 " SEL APPM "       SELECT GOAL MOVE:OPR .> MOVE:OPR - SELECT
M4 " SEL APPF1 "      SELECT GOAL FORM:OPR - FORM2INPUT:OPR .> FORM:OPR -
  SELECT
M5 " SEL APPF2 "      SELECT GOAL FORM:OPR FORM2INPUT:OPR .> FORM2INPUT:OPR -
  SELECT

      % MODULE TRANSF, EXCEPT M21 AND M23E ARE MATCH %

M20 " TRANSF:G "       TRANSF GOAL .> MATCH TRANSF
M20S " SUC TRANS "      TRANSF GOAL .> GOAL - TRANSF
M21 " MATCH RESULT "     MATCH .> EVAL TRANSF - MATCH
M22 " MATCH VAL "       TRANSF EVAL .> TRANSF - EVAL
M23 " MATCH FIN "       TRANSF .> MATCH TRANSF
M23E " ERASE MD "      MATCH .> - MATCH
M24 " COMP DIFFIC "      TRANSF .> TRANSF GOAL
M24E " ERS MVAL "      TRANSF .> - TRANSF
M24F " ERS MVAL- "      TRANSF .> - TRANSF
M24N " ERS MVAL SV- "    TRANSF .> TRANSF
M24S " ERS MVAL SV "     TRANSF .> TRANSF GOAL
M25 " SUC DESCR "       TRANSF GOAL OBJECT .> GOAL - TRANSF

M26 " NEW REDUCE "      GOAL .> FILE EVAL GOAL
M27 " NEW REDUCE "      GOAL .> FILE EVAL GOAL

      % MODULE REDUCE %

M30 " REDUCE:G "       REDUCE GOAL .> APPLY REDUCE
M31 " SEL OP "   REDUCE APPLY - SET MOVE:OPR .> GENRT GOAL - REDUCE - APPLY
M32 " SEL OP SET "      REDUCE APPLY SET MOVE:OPR .> GENRT GOAL - REDUCE -
  APPLY
M33 " SEL OP FORM "      REDUCE APPLY - SET FORM:OPR GOAL .> APPLY REDUCE
M33S " SEL FORM SET "    REDUCE APPLY SET MOVE:OPR FORM:OPR GOAL .> APPLY REDUCE

      % MODULE GENRT %

M34 " SEL DES ASG "     GENRT MOVE:OPR .> GENRT APPLY GOAL
M34A " SEL DES ASG ARB "       GENRT - MOVE:OPR MOVE:OPR COMPON VARBL .>
  LOCPROG FILE GENRT VARBL APPLY GOAL
M34M " SEL DES ASG# "    GENRT MOVE:OPR .> GENRT APPLY GOAL
M34N " SEL DES ASG# "    GENRT MOVE:OPR .> GENRT APPLY GOAL
M34R " ERS SEL "       GENRT - MOVE:OPR - COMPON - VARBL .> GENRT
M35 " GET COMP. "      GENRT LOCPROG .> GENRT
M35G " GET COMP "      GENRT - LOCPROG .> LOCPROG GENRT
M36 " GEN DES ASG "     GENRT COMPON LOCPROG VARBL .> GENRT FILE VARBL
M36A " GEN DES ASG ARB "       GENRT COMPON MOVE:OPR VARBL .> LOCPROG FILE
  GENRT VARBL
```

```
M37 " ERS L:C "        LOC:PROG - GENRT -> - LOC:PROG
M37R " ERS L:C RE-AS "  LOC:PROG GENRT -> GENRT - LOC:PROG
M37S " ERS L:C RE-AS "  LOC:PROG GENRT -> GENRT - LOC:PROG
M38 " GEN DES ASG* "    GENRT COMPON VARBL LOC:PROG -> LOC:PROG FILE GENRT
  VARBL
M38F " GEN ASG* - "     GENRT COMPON VARBL - LOC:PROG GOAL VARBL FILE ->
  LOC:PROG - GENRT - GOAL - VARBL - FILE
M39 " GEN DES ASG*- "   GENRT - COMPON - VARBL -> LOC:PROG - GENRT

      % MODULE APPLY %

M40 " TRY APPLY "        APPLY GOAL - GENRT -> APPLY - GOAL
M40H " TRY APPLY MULT "         APPLY GOAL -> APPLY
M40R " RE-AS FEASASG "   GOAL GENRT APPLY -> GENRT APPLY
M40U " UN-HOLD TRYAPP "         APPLY -> APPLY
M41 " APPLY SUC "        APPLY -> FILE GOAL - APPLY
M42 " EVAL OP DIFFR "     APPLY -> EVAL APPLY
M43 " DIFFR DIFFIC* "     APPLY EVAL -> APPLY - EVAL
M43F " DIFFR DIFFIC1 "    APPLY EVAL -> APPLY - EVAL
M43L " DIFFR DIFFIC- "    APPLY EVAL -> APPLY - EVAL
M44 " APPLY FAIL "        APPLY -> APPLY
M44E " APPLY ERASE "      APPLY VARBL -> - APPLY - VARBL
M45 " SEL DIFFR "        APPLY - GOAL -> APPLY
M45I " SEL IDENT "        APPLY - GOAL -> APPLY
M46 " NO DIFFR "         APPLY -> GOAL - APPLY
M46T " NO DIFFR TA* "     APPLY - GOAL -> GOAL - APPLY
M47 " TRYAPP RES "        APPLY GOAL -> FILE EVAL GOAL - APPLY

      % MODULE REDUCE AGAIN %

M48 " RETRY ASG "        REDUCE GOAL -> APPLY - REDUCE - GOAL
M49 " RETRY OPD "        REDUCE GOAL -> APPLY - REDUCE - GOAL
M49N " RETRY NOTHING "   REDUCE GOAL -> REDUCE - GOAL

      % MODULE MOVE:OPR %

M50 " MOVE OP "          MOVE:OPR - GOAL GOAL -> GOAL - MOVE:OPR
M51 " NEW RED APP "       GOAL -> FILE EVAL GOAL
M52 " MOVE OP - DIF "     MOVE:OPR - GOAL GOAL -> GENRT APPLY - MOVE:OPR
M57 " TRYAPP RES "        APPLY GOAL -> GOAL - APPLY
M58 " RETRY ASG "        MOVE:OPR GOAL -> APPLY - MOVE:OPR - GOAL
M59 " RETRY OPD "        MOVE:OPR GOAL -> APPLY - MOVE:OPR - GOAL

      % MODULE MATCH AGAIN %

K0 " MATCH:DIFF TOP "    MATCH OBJECT -> MATCH
K1 " MD * "      MATCH NODE - OBJECT -> MATCH
K3 " MD BAD VAL "        MATCH NODE -> MATCH
K4 " MD UNDEF N1 "       MATCH NODE -> MATCH
K5 " MD UNDEF N2 "       MATCH NODE -> MATCH
K6 " MD UNDEF V1 "       MATCH NODE -> MATCH
K7 " MD UNDEF V2 "       MATCH NODE -> MATCH
K8 " LOC EXTR "         MATCH NODE - OBJECT -> MATCH DIFFR
K9 " LOC EXTR TOP "       MATCH NODE OBJECT -> FILE DIFFR MATCH
K10 " MD RESULT F "      MATCH DIFFR -> MATCH - DIFFR
K11 " LOC EXTR TOP 1 "    MATCH NODE OBJECT -> MATCH DIFFR

      % MODULE ADD %

T1 " ADD LINK1 "         ADD OBJECT - NODE -> ADD NODE
T2 " ADD LINK 1- "       ADD OBJECT NODE -> ADD
T3 " ADD LINK N "        ADD - NODE -> ADD NODE
T4 " ADD LINK N- "       ADD NODE -> ADD
T5 " ADD LINK V "        ADD -> NODE - ADD
T6 " ADD LINK VT "        ADD OBJECT -> NODE - ADD

      % MODULE REMOVE %

T10 " REM LINK ALL TOP "        REMOVE OBJECT NODE -> REMOVE - NODE
T11 " REM LINK SPEC TOP "       REMOVE OBJECT NODE -> REMOVE
T12 " REM LINK ALL ARB "        REMOVE NODE -> REMOVE - NODE
T13 " REM LINK VAL "     REMOVE NODE -> - REMOVE - NODE
T14 " REM LINK ARB C "   REMOVE NODE -> REMOVE

      % MODULE INCR %

T20 " INCR LINK 1 "      INCR OBJECT NODE -> INCR
T21 " INCR LINK N "      INCR NODE -> INCR
T22 " INCR LINK V "      INCR NODE -> - INCR - NODE NODE
T23 " INCR LINK VT "      INCR OBJECT NODE -> - INCR - NODE NODE

      % MODULE DECR %

T30 " DECR LINK 1 "     DECR OBJECT NODE -> DECR
T31 " DECR LINK N "     DECR NODE -> DECR
T32 " DECR LINK V "     DECR NODE -> - DECR - NODE NODE
T33 " DECR LINK VT "     DECR OBJECT NODE -> - DECR - NODE NODE

      % MODULE COPY %

T40 " COPY LINK 1 "     COPY OBJECT NODE -> COPY
T41 " COPY LINK N "     COPY NODE -> COPY
T42 " COPY LINK V "     COPY NODE -> NODE - COPY
T43 " COPY LINK T "     COPY OBJECT NODE -> NODE - COPY

C1 " COPY OBJ TOP "     COPY OBJECT -> COPY OBJECT
C2 " COPY OBJ N "      COPY NODE -> COPY NODE
C3 " COPY OBJ NV "      COPY NODE -> NODE - COPY
C4 " COPY OBJ - "      COPY - NODE - OBJECT -> - COPY

      % MODULE EVAL AGAIN %

D1 " DIFFR EVAL "       EVAL -> APPLY EVAL
D2 " DIFFR EVAL R1 "     EVAL APPLY -> EVAL - APPLY
D3 " DIFFR EVAL R2 "     EVAL -> EVAL
D4 " DIFFR EVAL R2* "     EVAL -> EVAL
D5 " DIFFR EVAL R2T "     EVAL -> EVAL

      % MODULE TRACE %

V1 " TRANSF "    TRACE GOAL -> TRACE GOAL
V2 " OBJ "      TRACE OBJECT -> TRACE
V3 " TRANSF "    TRACE GOAL -> TRACE GOAL
V4 " APPLY "     TRACE GOAL -> TRACE GOAL
V5 " APPLY "     TRACE GOAL -> TRACE GOAL
V6 " REDUCE "    TRACE GOAL OBJECT -> TRACE GOAL
V7 " ASG 1 "     TRACE VARBL -> TRACE
V8 " ASG 2 "     TRACE VARBL -> TRACE
V9 " ASG 3 "     TRACE VARBL -> TRACE

      % MODULE OBJECT *EXTREPR %

X1 " EXT REPR "        OBJECT NODE -> NODE - OBJECT
X2 " DESC "     NODE -> NODE
X3 " BOTT "     NODE -> NODE
X4 " BOTT NIL "        NODE -> NODE
X5 " ASC "      NODE -> NODE
X6 " TOP "      NODE OBJECT TRACE -> OBJECT - NODE
```

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 AFOSR - TR - 77 - 0330 - Vol - 1 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 PRODUCTION SYSTEMS AS A PROGRAMMING LANGUAGE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS. Volume I.	5. TYPE OF REPORT & PERIOD COVERED 9 Interim Rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Michael D. Rychener	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209	12. REPORT DATE 11 December 1976
	13. NUMBER OF PAGES 503
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of S ientific Research (NM) Bolling AFB, DC 20332 12 153p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

16 2304 17 A2

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT This thesis develops a system architecture for artificial intelligence (AI), called production systems (PSs). Each production is a simple condition-action rule, with conditions stated on a global Working Memory and actions consisting primarily of simple modifications to that memory. Actions can also consist of forming new productions. PSs have been applied to a limited extent in computer science and to a somewhat larger extent to specialized studies in AI. They are used in cognitive psychology to model human intellectual capabilities at a detailed level. With AI research tending toward larger systems with greater flexibility requirements, PSs are promising as candidates for the primary

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED

403081 SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

knowledge encoding medium, but certain questions and problems with PSs have been raised. The questions revolve around the practical feasibity of PSs for building large systems in a diversity of task domains, the preservation of desirable PS properties when they are applied to much larger systems than previously, and the specific advantages and disadvantages of PS architectural features.

This thesis seeks answers to such questions by constructing PSs to perform the following tasks, all of which have been developed by past AI research: extracting equations from typical high school algebra story problems (Bobrow's STUDENT); learning lists of nonsense syllable pairs (Feigenbaum's EPAM); solving a variety of puzzle tasks using a single set of general methods and processes (Newell, Shaw, Simon and Ernst's GPS); playing a simple class of chess endgames (Perdue and Berliner); discoursing in natural language about a toy blocks scene (Moran's mini-linguistic system); and solving toy blocks manipulation problems (Winograd's SHRDLU system). Each implementation is analyzed to bring out PS characteristics.

Evaluations of PSs as a programming language are made according to the traits: practical feasibility, style, degree of theory-boundness, power and overhead of expression, productivity, efficiency, architectural flexibility, and level. A taxonomy of control is presented, and measures of frequencies of usages in the PSs of various forms of control in that taxonomy are used to support the discussion of power and overhead of expression. The actual PSs are able to effectively exploit PS power in the particular areas of selections and iterations. Specific features of the particular language design used here are central to the capabilities discussed. A taxonomy of representation is developed, to provide a basis for adding openness to the PSs, replacing ad hoc internal naming conventions, and to allow measurement of the modularity of PSs, making interdependencies of various parts more examinable. The taxonomy of representation is applied to one of the larger PS programs with the finding that the split between inter-module assumptions and intra-module assumptions is roughly an order of magnitude, approximately the form of a nearly decomposable system.

PSs are found to be effective and advantageous for the programming constructs typical of AI systems. They have particular advantages in style, conciseness, and architectural flexibility. Major successes can be expected in applying PSs to large-scale understanding systems of the sort currently being explored. They are particularly useful in domains where system knowledge must grow dynamically through interaction with humans and with a task environment, but without the expense of analysis of how each new piece of knowledge must fit into existing structure. Their diversity of application and their problem-solving capabilities, both of which are deemed essential to building understanding systems, have been adequately demonstrated by this thesis.